Docker Fundamentals for Linux Exercises

Windows Users: Please note that in all exercises we will use Unix style paths using forward slashes (‘/‘) instead of backslashes (‘\‘). On Windows you can work directly with such paths by either using a **Bash** terminal or a **Powershell** terminal. Powershell can work with both Windows and Unix style paths.

Be aware that copy-pasting of commands or code snippets from this PDF may apply changes to some characters e.g. quotes, tabs, which may leads to errors. Please consider typing suggested commands and code snippets in case you encounter any issues.

Contents

Exercises

1 Running & Inspecting Containers

By the end of this exercise, you should be able to:

- Start a container
- List running and stopped containers

1.1 Running Containers

1. First, let's start a container, and observe the output:

```
[centos@node-0 ~]$ docker container run centos:7 echo "hello world"

Unable to find image 'centos:7' locally
7: Pulling from library/centos
256b176beaff: Pull complete
Digest: sha256:6f6d986d425aeabdc3a02cb61c02abb2e78e57357e92417d6d58332856024faf
Status: Downloaded newer image for centos:7
hello world
```

   The `centos:7` part of the command indicates the *image* we want to use to define this container; it defines a private filesystem for the container. `echo "hello world"` is the process we want to execute inside the kernel namespaces created when we use `docker container run`.

   Since we've never used the `centos:7` image before, first Docker downloads it, and then runs our `echo "hello world"` process inside a contianer, sending the STDOUT stream of that process to our terminal by default.

2. Now create another container from the same image, and run a different process inside of it:

```
[centos@node-0 ~]$ docker container run centos:7 ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 14:28 ?        00:00:00 ps -ef
```

   No download this time, and we can see that our containerized process (`ps -ef` in this case) is PID 1 inside the container.

3. Try doing `ps -ef` at the host prompt and see what process is PID 1 here.

1.2 Listing Containers

1. Try listing all your currently running containers:

```
[centos@node-0 ~]$ docker container ls
```

   There's nothing listed, since the containers you ran executed a single command, and shut down when finished.

2. List stopped as well as running containers with the `-a` flag:

```
[centos@node-0 ~]$ docker container ls -a

CONTAINER ID  IMAGE      COMMAND                  CREATED              STATUS
a525daef85ab  centos:7   "ps -ef"                 About a minute ago   Exited (0) About a minute ago
db6aabba5157  centos:7   "echo 'hello world'"     3 minutes ago        Exited (0) 3 minutes ago
```


We can see our exited containers this time, with a time and exit code in the `STATUS` column.

> **Where did those names come from?** We truncated the above output table, but in yours you should also see a `NAMES` column with some funny names. All containers have names, which in most Docker CLI commands can be substituted for the container ID as we'll see in later exercises. By default, containers get a randomly generated name of the form `<adjective>_<scientist / technologist>`, but you can choose a name explicitly with the `--name` flag in `docker container run`.

3. Clean up all containers using this command:

```
[centos@node-0 ~]$ docker container rm -f $(docker container ls -aq)
```

Please discuss with your peers what the above command exactly does.

1.3 Conclusion

In this exercise you ran your first container using `docker container run`, and explored the importance of the PID 1 process in a container; this process is a member of the host's PID tree like any other, but is 'containerized' via tools like kernel namespaces, making this process and its children behave as if it was the root of a PID tree, with its own filesystem, mountpoints, and network stack. The PID 1 process in a container defines the lifecycle of the container itself; when one exits, so does the other.

2 Interactive Containers

By the end of this exercise, you should be able to:

- Launch an interactive shell in a new or existing container
- Run a child process inside a running container
- List containers using more options and filters

2.1 Writing to Containers

1. Create a container using the `centos:7` image, and connect to its bash shell in interactive mode using the `-i` flag (also the `-t` flag, to request a TTY connection):

```
[centos@node-0 ~]$ docker container run -it centos:7 bash
```

2. Explore your container's filesystem with `ls`, and then create a new file:

```
[root@2b8de2ffdf85 /]# ls -l
[root@2b8de2ffdf85 /]# echo 'Hello there...' > test.txt
[root@2b8de2ffdf85 /]# ls -l
```

3. Exit the connection to the container:

```
[root@2b8de2ffdf85 /]# exit
```

4. Run the same command as above to start a container in the same way:

```
[centos@node-0 ~]$ docker container run -it centos:7 bash
```

5. Try finding your `test.txt` file inside this new container; it is nowhere to be found. Exit this container for now in the same way you did above.

2.2 Reconnecting to Containers

1. We'd like to recover the information written to our container in the first example, but starting a new container didn't get us there; instead, we need to restart our original container, and reconnect to it. List all your stopped

containers:

```
[centos@node-0 ~]$ docker container ls -a

CONTAINER ID  IMAGE     COMMAND  CREATED              STATUS                          ...
cc19f7e9aa91  centos:7  "bash"   About a minute ago   Exited (0) About a minute ago  ...
2b8de2ffdf85  centos:7  "bash"   2 minutes ago        Exited (0) About a minute ago  ...
...
```

2. We can restart a container via the container ID listed in the first column. Use the container ID for the first `centos:7` container you created with `bash` as its command (see the `CREATED` column above to make sure you're choosing the *first* bash container you ran):

```
[centos@node-0 ~]$ docker container start <container ID>
[centos@node-0 ~]$ docker container ls

CONTAINER ID  IMAGE     COMMAND  CREATED         STATUS         ...
2b8de2ffdf85  centos:7  "bash"   5 minutes ago   Up 21 seconds  ...
```

   Your container status has changed from `Exited` to `Up`, via `docker container start`.

3. Run `ps -ef` inside the container you just restarted using Docker's `exec` command (`exec` runs the specified process as a child of the PID 1 process inside the container):

```
[centos@node-0 ~]$ docker container exec <container ID> ps -ef
```

   What process is PID 1 inside the container? Find the PID of that process on the host machine by using:

```
[centos@node-0 ~]$ docker container top <container ID>
```

4. Launch a bash shell in your running container with `docker container exec`:

```
[centos@node-0 ~]$ docker container exec -it <container ID> bash
```

5. List the contents of the container's filesystem again with `ls -l`; your `test.txt` should be where you left it. Exit the container again by typing `exit`.

2.3 Using Container Listing Options

1. In the last step, we saw how to get the short container ID of all our containers using `docker container ls -a`. Try adding the `--no-trunc` flag to see the entire container ID:

```
[centos@node-0 ~]$ docker container ls -a --no-trunc
```

   This long ID is the same as the string that is returned after starting a container with `docker container run`.

2. List only the container ID using the `-q` flag:

```
[centos@node-0 ~]$ docker container ls -a -q
```

3. List the last container to have been created using the `-l` flag:

```
[centos@node-0 ~]$ docker container ls -l
```

4. Finally, you can also filter results with the `--filter` flag; for example, try filtering by exit code:

```
[centos@node-0 ~]$ docker container ls -a --filter "exited=0"
```

   The output of this command will list the containers that have exited successfully.

5. Clean up with:

```
[centos@node-0 ~]$ docker container rm -f $(docker container ls -aq)
```


2.4 Conclusion

In this demo, you saw that files added to a container's filesystem do not get added to all containers created from the same image; changes to a container's filesystem are local to itself, and exist only in that particular container. You also learned how to restart a stopped Docker container using `docker container start`, how to run a command in a running container using `docker container exec`, and also saw some more options for listing containers via `docker container ls`.

3 Detached Containers and Logging

By the end of this exercise, you should be able to:

- Run a container detached from the terminal
- Fetch the logs of a container
- Attach a terminal to the STDOUT of a running container

3.1 Running a Container in the Background

1. First try running a container as usual; the STDOUT and STDERR streams from whatever is PID 1 inside the container are directed to the terminal:

```
[centos@node-0 ~]$ docker container run centos:7 ping 127.0.0.1 -c 2

PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.021 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.029 ms

--- 127.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1019ms
rtt min/avg/max/mdev = 0.021/0.025/0.029/0.004 ms
```

2. The same process can be run in the background with the `-d` flag:

```
[centos@node-0 ~]$ docker container run -d centos:7 ping 127.0.0.1

d5ef517cc113f36738005295066b271ae604e9552ce4070caffbacdc3893ae04
```

   This time, we only see the container's ID; its STDOUT isn't being sent to the terminal.

3. Use this second container's ID to inspect the logs it generated:

```
[centos@node-0 ~]$ docker container logs <container ID>
```

   These logs correspond to STDOUT and STDERR from the container's PID 1. Also note when using container IDs: you don't need to specify the entire ID. Just enough characters from the start of the ID to uniquely identify it, often just 2 or 3, is sufficient.

3.2 Attaching to Container Output

1. We can attach a terminal to a container's PID 1 output with the `attach` command; try it with the last container you made in the previous step:

```
[centos@node-0 ~]$ docker container attach <container ID>
```

2. We can leave attached mode by then pressing `CTRL+C`. After doing so, list your running containers; you should see that the container you attached to has been killed, since the `CTRL+C` issued killed PID 1 in the container, and therefore the container itself.

3. Try running the same thing in detached interactive mode:

```
[centos@node-0 ~]$ docker container run -d -it centos:7 ping 127.0.0.1
```

4. Attach to this container like you did the first one, but this time detach with `CTRL+P CTRL+Q` (sequential, not simultaneous), and list your running containers. In this case, the container should still be happily running in the background after detaching from it.

3.3 Using Logging Options

1. We saw previously how to read the entire log of a container's PID 1; we can also use a couple of flags to control what logs are displayed. `--tail n` limits the display to the last `n` lines; try it with the container that should be running from the last step:

```
[centos@node-0 ~]$ docker container logs --tail 5 <container ID>
```

   You should see the last 5 pings from this container.

2. We can also follow the logs as they are generated with `-f`:

```
[centos@node-0 ~]$ docker container logs -f <container ID>
```

   The container's logs get piped in real time to the terminal (`CTRL+C` to break out of following mode - note this doesn't kill the process like when we attached to it, since now we're tailing the logs, not attaching to the process).

3. Finally, try combining the tail and follow flags to begin following the logs from 10 lines back in history.

3.4 Conclusion

In this exercise, we saw our first detached containers. Almost all containers you ever run will be running in detached mode; you can use `container attach` to interact with their PID 1 processes, as well as `container logs` to fetch their logs. Note that both `attach` and `logs` interact with the PID 1 process only - if you launch child processes inside a container, it's up to you to manage their STDOUT and STDERR streams. Also, be careful when killing processes after attaching to a container; as we saw, it's easy to attach to a container and then kill it, by issuing a `CTRL+C` to the PID 1 process you've attached to.

4 Starting, Stopping, Inspecting and Deleting Containers

By the end of this exercise, you should be able to:

- Restart containers which have exited
- Distinguish between stopping and killing a container
- Fetch container metadata using `docker container inspect`
- Delete containers

4.1 Starting and Restarting Containers

1. Start by running a container in the background, and check that it's really running:

```
[centos@node-0 ~]$ docker container run -d centos:7 ping 8.8.8.8
[centos@node-0 ~]$ docker container ls
```

2. Stop the container using `docker container stop`, and check that the container is indeed stopped:

```
[centos@node-0 ~]$ docker container stop <container ID>
[centos@node-0 ~]$ docker container ls -a
```


Note that the `stop` command takes a few seconds to complete. `docker container stop` first sends a `SIGTERM` to the PID 1 process inside a container, asking it to shut down nicely; it then waits 10 seconds before sending a `SIGKILL` to kill it off, ready or not. The exit code you see (137 in this case) is the exit code returned by the PID 1 process (`ping`) upon being killed by one of these signals.

3. Start the container again with `docker container start`, and attach to it at the same time with the `-a` flag:

```
[centos@node-0 ~]$ docker container start -a <container ID>
```

As you saw previously, this brings the container from the `Exited` to the `Up` state; in this case, we're also attaching to the PID 1 process.

4. Detach and stop the container with `CTRL+C`, then restart the container without attaching and follow the logs starting from 10 lines previous.

5. Finally, stop the container with `docker container kill`:

```
[centos@node-0 ~]$ docker container kill <container ID>
```

Unlike `docker container stop`, `container kill` just sends the `SIGKILL` right away - no grace period.

4.2 Inspecting a Container

1. Start your ping container again, then inspect the container details using `docker container inspect`:

```
[centos@node-0 ~]$ docker container start <container ID>
[centos@node-0 ~]$ docker container inspect <container ID>
```

You get a JSON object describing the container's config, metadata and state.

2. Find the container's IP and long ID in the JSON output of `inspect`. If you know the key name of the property you're looking for, try piping to grep:

```
[centos@node-0 ~]$ docker container inspect <container ID> | grep IPAddress
```

The output should look similar to this:

```
"SecondaryIPAddresses": null,
"IPAddress": "<Your IP Address>"
```

3. Now try grepping for `Cmd`, the PID 1 command being run by this container. `grep`'s simple text search doesn't always return helpful results:

```
[centos@node-0 ~]$ docker container inspect <container ID> | grep Cmd

"Cmd": [
```

4. A more powerful way to filter this JSON is with the `--format` flag. Syntax follows Go's text/template package: http://golang.org/pkg/text/template/. For example, to find the `Cmd` value we tried to grep for above, instead try:

```
[centos@node-0 ~]$ docker container inspect --format='{{.Config.Cmd}}' <container ID>

[ping 8.8.8.8]
```

This time, we get a the value of the `Config.Cmd` key from the `inspect` JSON.

5. Keys nested in the JSON returned by `docker container inspect` can be chained together in this fashion. Try modifying this example to return the IP address you grepped for previously.

6. Finally, we can extract all the key/value pairs for a given object using the `json` function:

```
[centos@node-0 ~]$ docker container inspect --format='{{json .Config}}' <container ID>
```

Try adding `| jq` to this command to get the same output a little bit easier to read.

4.3 Deleting Containers

1. Start three containers in background mode, then stop the first one.
2. List only exited containers using the `--filter` flag we learned earlier, and the option `status=exited`.
3. Delete the container you stopped above with `docker container rm`, and do the same listing operation as above to confirm that it has been removed:

```
[centos@node-0 ~]$ docker container rm <container ID>
[centos@node-0 ~]$ docker container ls ...
```

4. Now do the same to one of the containers that's still running; notice `docker container rm` won't delete a container that's still running, unless we pass it the force flag `-f`. Delete the second container you started above:

```
[centos@node-0 ~]$ docker container rm -f <container ID>
```

5. Try using the `docker container ls` flags we learned previously to remove the last container that was run, or all stopped containers. Recall that you can pass the output of one shell command `cmd-A` into a variable of another command `cmd-B` with syntax like `cmd-B $(cmd-A)`.
6. When done, clean up any containers you may still have:

```
[centos@node-0 ~]$ docker container rm -f $(docker container ls -aq)
```

4.4 Conclusion

In this exercise, you explored the lifecycle of a container, particularly in terms of stopping and restarting containers. Keep in mind the behavior of `docker container stop`, which sends a `SIGTERM`, waits a grace period, and then sends a `SIGKILL` before forcing a container to stop; this two step process is designed to give your containers a chance to shut down 'nicely': dump their state to a log, finish a database transaction, or do whatever your application needs them to do in order to exit without causing additional problems. Make sure you bear this in mind when designing containerized software.

Also keep in mind the `docker container inspect` command we saw, for examining container metadata, state and config; this is often the first place to look when trying to troubleshoot a failed container.

5 Interactive Image Creation

By the end of this exercise, you should be able to:

- Capture a container's filesystem state as a new docker image
- Read and understand the output of `docker container diff`

5.1 Modifying a Container

1. Start a bash terminal in a CentOS container:

```
[centos@node-0 ~]$ docker container run -it centos:7 bash
```

2. Install a couple pieces of software in this container - there's nothing special about `wget`, any changes to the filesystem will do. Afterwards, exit the container:

```
[root@dfe86ed42be9 /]# yum install -y which wget
[root@dfe86ed42be9 /]# exit
```

3. Finally, try `docker container diff` to see what's changed about a container relative to its image; you'll need to get the container ID via `docker container ls -a` first:


```
[centos@node-0 ~]$ docker container ls -a
[centos@node-0 ~]$ docker container diff <container ID>

C /root
A /root/.bash_history
C /usr
C /usr/bin
A /usr/bin/gsoelim
...
```

Those `C`s at the beginning of each line stand for files `C`hanged, and `A` for `A`dded; lines that start with `D` indicate `D`eletions.

5.2 Capturing Container State as an Image

1. Installing `which` and `wget` in the last step wrote information to the container's read/write layer; now let's save that read/write layer as a new read-only image layer in order to create a new image that reflects our additions, via the `docker container commit`:

   ```
   [centos@node-0 ~]$ docker container commit <container ID> myapp:1.0
   ```

2. Check that you can see your new image by listing all your images:

   ```
   [centos@node-0 ~]$ docker image ls

   REPOSITORY  TAG   IMAGE ID       CREATED         SIZE
   myapp       1.0   34f97e0b087b   8 seconds ago   300MB
   centos      7     5182e96772bf   44 hours ago    200MB
   ```

3. Create a container running bash using your new image, and check that vim and wget are installed:

   ```
   [centos@node-0 ~]$ docker container run -it myapp:1.0 bash
   [root@2ecb80c76853 /]# which wget
   ```

   The `which` commands should show the path to the specified executable, indicating they have been installed in the image.

5.3 Conclusion

In this exercise, you saw how to inspect the contents of a container's read / write layer with `docker container diff`, and commit those changes to a new image layer with `docker container commit`. Committing a container as an image in this fashion can be useful when developing an environment inside a container, when you want to capture that environment for reproduction elsewhere.

6 Creating Images with Dockerfiles (1/2)

By the end of this exercise, you should be able to:

- Write a Dockerfile using the `FROM` and `RUN` commands
- Build an image from a Dockerfile
- Anticipate which image layers will be fetched from the cache at build time
- Fetch build history for an image

6.1 Writing and Building a Dockerfile

1. Create a folder called `myimage`, and a text file called `Dockerfile` within that folder. In `Dockerfile`, include the following instructions:

```
FROM centos:7

RUN yum update -y
RUN yum install -y wget
```

   This serves as a recipe for an image based on `centos:7`, that has all its default packages updated and `wget` installed on top.

2. Build your image with the `build` command. Don't miss the `.` at the end; that's the path to your `Dockerfile`. Since we're currently in the directory `myimage` which contains it, the path is just `.` (here).

```
[centos@node-0 myimage]$ docker image build -t myimage .
```

   You'll see a long build output - we'll go through the meaning of this output in a demo later. For now, everything is good if it ends with `Successfully tagged myimage:latest`.

3. Verify that your new image exists with `docker image ls`, then use it to run a container and `wget` something from within that container, just to confirm that everything worked as expected:

```
[centos@node-0 myimage]$ docker container run -it myimage bash
[root@1d86d4093cce /]# wget example.com
[root@1d86d4093cce /]# cat index.html
[root@1d86d4093cce /]# exit
```

   You should see the HTML from example.com, downloaded by `wget` from within your container.

4. It's also possible to pipe a Dockerfile in from STDIN; try rebuilding your image with the following:

```
[centos@node-0 myimage]$ cat Dockerfile | docker image build -t myimage -f - .
```

   (This is useful when reading a Dockerfile from a remote location with `curl`, for example).

6.2 Using the Build Cache

In the previous step, the second time you built your image should have completed immediately, with each step save the first reporting `using cache`. Cached build steps will be used until a change in the Dockerfile is found by the builder.

1. Open your Dockerfile and add another `RUN` step at the end to install `vim`:

```
FROM centos:7

RUN yum update -y
RUN yum install -y wget
RUN yum install -y vim
```

2. Build the image again as above; which steps is the cache used for?
3. Build the image again; which steps use the cache this time?
4. Swap the order of the two `RUN` commands for installing `wget` and `vim` in the Dockerfile:

```
FROM centos:7

RUN yum update -y
RUN yum install -y vim
RUN yum install -y wget
```

   Build one last time. Which steps are cached this time?

6.3 Using the history Command

1. The docker image history command allows us to inspect the build cache history of an image. Try it with your new image:

```
[centos@node-0 myimage]$ docker image history myimage:latest

IMAGE          CREATED          CREATED BY                                      SIZE
f2e85c162453   8 seconds ago    /bin/sh -c yum install -y wget                  87.2MB
93385ea67464   12 seconds ago   /bin/sh -c yum install -y vim                   142MB
27ad488e6b79   3 minutes ago    /bin/sh -c yum update -y                        86.5MB
5182e96772bf   44 hours ago     /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      44 hours ago     /bin/sh -c #(nop)  LABEL org.label-schema....   0B
<missing>      44 hours ago     /bin/sh -c #(nop) ADD file:6340c690b08865d...   200MB
```

   Note the image id of the layer built for the yum update command.

2. Replace the two RUN commands that installed wget and vim with a single command:

```
...
RUN yum install -y wget vim
```

3. Build the image again, and run docker image history on this new image. How has the history changed?

6.4 Conclusion

In this exercise, we've seen how to write a basic Dockerfile using FROM and RUN commands, some basics of how image caching works, and seen the docker image history command. Using the build cache effectively is crucial for images that involve lengthy compile or download steps; in general, moving commands that change frequently as late as possible in the Dockerfile will minimize build times. We'll see some more specific advice on this later in this lesson.

7 Creating Images with Dockerfiles (2/2)

By the end of this exercise, you should be able to:

- Define a default process for an image to containerize by using the ENTRYPOINT or CMD Dockerfile commands
- Understand the differences and interactions between ENTRYPOINT and CMD

7.1 Setting Default Commands

1. Add the following line to your Dockerfile from the last problem, at the bottom:

```
...
CMD ["ping", "127.0.0.1", "-c", "5"]
```

   This sets ping as the default command to run in a container created from this image, and also sets some parameters for that command.

2. Rebuild your image:

```
[centos@node-0 myimage]$ docker image build -t myimage .
```

3. Run a container from your new image with no command provided:

```
[centos@node-0 myimage]$ docker container run myimage
```

   You should see the command provided by the CMD parameter in the Dockerfile running.

4. Try explicitly providing a command when running a container:


```
[centos@node-0 myimage]$ docker container run myimage echo "hello world"
```

Providing a command in `docker container run` overrides the command defined by `CMD`.

5. Replace the `CMD` instruction in your Dockerfile with an `ENTRYPOINT`:

```
...
ENTRYPOINT ["ping"]
```

6. Build the image and use it to run a container with no process arguments:

```
[centos@node-0 myimage]$ docker image build -t myimage .
[centos@node-0 myimage]$ docker container run myimage
```

You'll get an error. What went wrong?

7. Try running with an argument after the image name:

```
[centos@node-0 myimage]$ docker container run myimage 127.0.0.1
```

You should see a successful ping output. Tokens provided after an image name are sent as arguments to the command specified by `ENTRYPOINT`.

7.2 Combining Default Commands and Options

1. Open your Dockerfile and modify the `ENTRYPOINT` instruction to include 2 arguments for the ping command:

```
...
ENTRYPOINT ["ping", "-c", "3"]
```

2. If `CMD` and `ENTRYPOINT` are both specified in a Dockerfile, tokens listed in `CMD` are used as default parameters for the `ENTRYPOINT` command. Add a `CMD` with a default IP to ping:

```
...
CMD ["127.0.0.1"]
```

3. Build the image and run a container with the defaults:

```
[centos@node-0 myimage]$ docker image build -t myimage .
[centos@node-0 myimage]$ docker container run myimage
```

You should see it pinging the default IP, `127.0.0.1`.

4. Run another container with a custom IP argument:

```
[centos@node-0 myimage]$ docker container run myimage 8.8.8.8
```

This time, you should see a ping to `8.8.8.8`. Explain the difference in behavior between these two last containers.

7.3 Conclusion

In this exercise, we encountered the Dockerfile commands `CMD` and `ENTRYPOINT`. These are useful for defining the default process to run as PID 1 inside the container right in the Dockerfile, making our containers more like executables and adding clarity to exactly what process was meant to run in a given image's containers.

8 Multi-Stage Builds

By the end of this exercise, you should be able to:

- Write a Dockerfile that describes multiple images, which can copy files from one image to the next.
- Enable BuildKit for faster build times

8.1 Defining a multi-stage build

1. Make a fresh folder `~/multi` to do this exercise in, and `cd` into it.
2. Add a file `hello.c` to the `multi` folder containing **Hello World** in C:

```
#include <stdio.h>

int main (void)
{
    printf ("Hello, world!\n");
    return 0;
}
```

3. Try compiling and running this right on the host OS:

```
[centos@node-0 multi]$ gcc -Wall hello.c -o hello
[centos@node-0 multi]$ ./hello
```

4. Now let's Dockerize our hello world application. Add a `Dockerfile` to the `multi` folder with this content:

```
FROM alpine:3.5
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
CMD /app/bin/hello
```

5. Build the image and observe its size:

```
[centos@node-0 multi]$ docker image build -t my-app-large .
[centos@node-0 multi]$ docker image ls | grep my-app-large

REPOSITORY     TAG      IMAGE ID       CREATED         SIZE
my-app-large   latest   a7d0c6fe0849   3 seconds ago   189MB
```

6. Test the image to confirm it actually works:

```
[centos@node-0 multi]$ docker container run my-app-large
```

   It should print "hello world" in the console.

7. Update your Dockerfile to use an `AS` clause on the first line, and add a second stanza describing a second build stage:

```
FROM alpine:3.5 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
COPY hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello

FROM alpine:3.5
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

8. Build the image again and compare the size with the previous version:


```
[centos@node-0 multi]$ docker image build -t my-app-small .
[centos@node-0 multi]$ docker image ls | grep 'my-app-'

REPOSITORY     TAG      IMAGE ID       CREATED              SIZE
my-app-small   latest   f49ec3971aa6   6 seconds ago        4.01MB
my-app-large   latest   a7d0c6fe0849   About a minute ago   189MB
```

As expected, the size of the multi-stage build is much smaller than the large one since it does not contain the Alpine SDK.

9. Finally, make sure the app actually works:

```
[centos@node-0 multi]$ docker container run --rm my-app-small
```

You should get the expected 'Hello, World!' output from the container with just the required executable.

8.2 Building Intermediate Images

In the previous step, we took our compiled executable from the first build stage, but that image wasn't tagged as a regular image we can use to start containers with; only the final `FROM` statement generated a tagged image. In this step, we'll see how to persist whichever build stage we like.

1. Build an image from the `build` stage in your Dockerfile using the `--target` flag:

```
[centos@node-0 multi]$ docker image build -t my-build-stage --target build .
```

Notice all its layers are pulled from the cache; even though the build stage wasn't tagged originally, its layers are nevertheless persisted in the cache.

2. Run a container from this image and make sure it yields the expected result:

```
[centos@node-0 multi]$ docker container run -it --rm my-build-stage /app/bin/hello
```

3. List your images again to see the size of `my-build-stage` compared to the small version of the app.

8.3 Optional: Enabling BuildKit

In addition to the default builder, BuildKit can be enabled to take advantages of some optimizations of the build process.

1. Turn on BuildKit:

```
[centos@node-0 multi]$ export DOCKER_BUILDKIT=1
```

2. Add an `AS` label to the final stage of your Dockerfile (this is not strictly necessary, but will make the output in the next step easier to understand):

```
...

FROM alpine:3.5 AS prod
RUN apk update
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

3. Re-build `my-app-small`, without the cache:

```
[centos@node-0 multi]$ docker image build --no-cache -t my-app-small-bk .

[+] Building 15.5s (14/14) FINISHED
 => [internal] load Dockerfile
 => => transferring dockerfile: 97B
 => [internal] load .dockerignore
```



```
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/alpine:3.5
=> CACHED [prod 1/3] FROM docker.io/library/alpine:3.5
=> [internal] load build context
=> => transferring context: 87B
=> CACHED [internal] helper image for file operations
=> [build 2/6] RUN apk update &&     apk add --update alpine-sdk
=> [prod 2/3] RUN apk update
=> [build 3/6] RUN mkdir /app
=> [build 4/6] COPY hello.c /app
=> [build 5/6] RUN mkdir bin
=> [build 6/6] RUN gcc -Wall hello.c -o bin/hello
=> [prod 3/3] COPY --from=build /app/bin/hello /app/hello
=> exporting to image
=> => exporting layers
=> => writing image sha256:22de288280df724625ec9be4b0e05086cd6c8d9327be9bcb1357b8c63df77847
=> => naming to docker.io/library/my-app-small-bk
```

Notice the lines marked like `[prod 2/3]` and `[build 4/6]`: `prod` and `build` in this context are the `AS` labels you applied to the `FROM` lines in each stage of your build in the Dockerfile; from the above output, you can see that the build stages were built in parallel. Every step of the final image was completed while the build environment image was being created; the `prod` environment image creation was only blocked at the `COPY` instruction since it required a file from the completed `build` image.

4. Comment out the `COPY` instruction in the `prod` image definition in your Dockerfile, and rebuild; the `build` image is skipped. BuildKit recognized that the `build` stage was not necessary for the image being built, and skipped it.

5. Turn off BuildKit:

```
[centos@node-0 multi]$ export DOCKER_BUILDKIT=0
```

8.4 Conclusion

In this exercise, you created a Dockerfile defining multiple build stages. Being able to take artifacts like compiled binaries from one image and insert them into another allows you to create very lightweight images that do not include developer tools or other unnecessary components in your production-ready images, just like how you currently probably have separate build and run environments for your software. This will result in containers that start faster, and are less vulnerable to attack.

9 Managing Images

By the end of this exercise, you should be able to:

- Rename and retag an image
- Push and pull images from the public registry
- Delete image tags and image layers, and understand the difference between the two operations

9.1 Making an Account on Docker's Hosted Registry

1. If you don't have one already, head over to https://hub.docker.com and make an account.

   For the rest of this workshop, `<Docker ID>` refers to the username you chose for this account.

9.2 Tagging and Listing Images

1. Download the `centos:7` image from Docker Hub:

   ```
   [centos@node-0 ~]$ docker image pull centos:7
   ```

2. Make a new tag of this image:

   ```
   [centos@node-0 ~]$ docker image tag centos:7 my-centos:dev
   ```

   Note no new image has been created; `my-centos:dev` is just a pointer pointing to the same image as `centos:7`.

3. List your images:

   ```
   [centos@node-0 ~]$ docker image ls
   ```

   You should have `centos:7` and `my-centos:dev` both listed, but they ought to have the same hash under image ID, since they're actually the same image.

9.3 Sharing Images on Docker Hub

1. Push your image to Docker Hub:

   ```
   [centos@node-0 ~]$ docker image push my-centos:dev
   ```

   You should get a `denied: requested access to the resource is denied` error.

2. Login by doing `docker login`, and try pushing again. The push fails again because we haven't namespaced our image correctly for distribution on Docker Hub; all images you want to share on Docker Hub must be named like `<Docker ID>/<repo name>[:<optional tag>]`.

3. Retag your image to be namespaced properly, and push again:

   ```
   [centos@node-0 ~]$ docker image tag my-centos:dev <Docker ID>/my-centos:dev
   [centos@node-0 ~]$ docker image push <Docker ID>/my-centos:dev
   ```

4. Search Docker Hub for your new `<Docker ID>/my-centos` repo, and confirm that you can see the `:dev` tag therein.

5. Next, write a Dockerfile that uses `<Docker ID>/my-centos:dev` as its base image, and installs any application you like on top of that. Build the image, and simultaneously tag it as `:1.0`:

   ```
   [centos@node-0 ~]$ docker image build -t <Docker ID>/my-centos:1.0 .
   ```

6. Push your `:1.0` tag to Docker Hub, and confirm you can see it in the appropriate repository.

7. Finally, list the images currently on your node with `docker image ls`. You should still have the version of your image that wasn't namespaced with your Docker Hub user name; delete this using `docker image rm`:

   ```
   [centos@node-0 ~]$ docker image rm my-centos:dev
   ```

   Only the tag gets deleted, not the actual image. The image layers are still referenced by another tag.

9.4 Conclusion

In this exercise, we praciced tagging images and exchanging them on the public registry. The namespacing rules for images on registries are *mandatory*: user-generated images to be exchanged on the public registry must be named like `<Docker ID>/<repo name>[:<optional tag>]`; official images in the Docker registry just have the repo name and tag.

Also note that as we saw when building images, image names and tags are just pointers; deleting an image with `docker image rm` just deletes that pointer if the corresponding image layers are still being referenced by another such pointer. Only when the last pointer is deleted are the image layers actually destroyed by `docker image rm`.

10 Database Volumes

By the end of this exercise, you should be able to:

- Provide a docker volume as a database backing to Postgres
- Make one Postgres container's database available to other Postgres containers

10.1 Launching Postgres

1. Download a postgres image, and look at its history to determine its default volume usage:

```
[centos@node-0 ~]$ docker image pull postgres:9-alpine
[centos@node-0 ~]$ docker image inspect postgres:9-alpine

...
"Volumes": {
    "/var/lib/postgresql/data": {}
},
...
```

   You should see a `Volumes` block like the above, indicating that those paths in the container filesystem will get volumes automatically mounted to them when a container is started based on this image.

2. Set up a running instance of this postgres container:

```
[centos@node-0 ~]$ docker container run --name some-postgres \
    -v db_backing:/var/lib/postgresql/data \
    -d postgres:9-alpine
```

   Notice the explicit volume mount, `-v db_backing:/var/lib/postgresql/data`; if we hadn't done this, a randomly named volume would have been mounted to the container's `/var/lib/postgresql/data`. Naming the volume explicitly is a best practice that will become useful when we start mounting this volume in multiple containers.

10.2 Writing to the Database

1. The `psql` command line interface to postgres comes packaged with the postgres image; spawn it as a child process in your postgres container interactively, to create a postgres terminal:

```
[centos@node-0 ~]$ docker container exec \
    -it some-postgres psql -U postgres
```

2. Create an arbitrary table in the database:

```
postgres=# CREATE TABLE CATICECREAM(COAT TEXT, ICECREAM TEXT);
postgres=# INSERT INTO CATICECREAM VALUES('calico', 'strawberry');
postgres=# INSERT INTO CATICECREAM VALUES('tabby', 'lemon');
```

   Double check you created the table you expected, and then quit this container:

```
postgres=# SELECT * FROM CATICECREAM;

  coat   |  icecream
---------+------------
 calico | strawberry
 tabby   | lemon
(2 rows)

postgres=# \q
```


3. Delete the postgres container:

```
[centos@node-0 ~]$ docker container rm -f some-postgres
```

4. Create a new postgres container, mounting the db_backing volume just like last time:

```
[centos@node-0 ~]$ docker container run \
    --name some-postgres \
    -v db_backing:/var/lib/postgresql/data \
    -d postgres:9-alpine
```

5. Reconnect a psql interface to your database, also like before:

```
[centos@node-0 ~]$ docker container exec \
    -it some-postgres psql -U postgres
```

6. List the contents of the CATICECREAM table:

```
postgres=# SELECT * FROM CATICECREAM;
```

   The contents of the database have survived the deletion and recreation of the database container; this would not have been true if the database was keeping its data in the writable container layer. As above, use \q to quit from the postgres prompt.

10.3 Running Multiple Database Containers

1. Create another postgres runtime, mounting the same backing volume:

```
[centos@node-0 ~]$ docker container run \
    --name another-postgres \
    -v db_backing:/var/lib/postgresql/data \
    -d postgres:9-alpine
```

2. Create another postgres interactive prompt, pointing at this new postgres container:

```
[centos@node-0 ~]$ docker container exec \
    -it another-postgres psql -U postgres
```

3. List the contents of the database one last time, again with SELECT * FROM CATICECREAM;. The database is readable exactly as it is from the other running database runtime, from this new postgres container.

4. Clean up by removing all your containers and deleting your postgres volume:

```
[centos@node-0 ~]$ docker container rm -f $(docker container ls -aq)
[centos@node-0 ~]$ docker volume rm db_backing
```

10.4 Conclusion

Whenever data needs to live longer than the lifecycle of a container, it should be pushed out to a volume outside the container's filesystem; numerous popular databases are containerized using this pattern. In addition to making sure data survives container deletion, this pattern allows us to share data among multiple containers, so multiple database instances can access the same underlying data.

11 Introduction to Container Networking

By the end of this exercise, you should be able to:

- Create docker bridge networks and attach containers to them
- Design networks of containers that can successfully resolve each other via DNS and reach each other across a Docker software defined network.

11.1 Inspecting the Default Bridge

1. See what networks are present on your host:

```
[centos@node-1 ~]$ docker network ls
```

   You should have entries for `host`, `none`, and `bridge`.

2. Find some metadata about the default `bridge` network:

```
[centos@node-1 ~]$ docker network inspect bridge
```

   Note especially the private subnet assigned by Docker's IPAM driver to this network. The first IP in this range is used as the network's gateway, and the rest will be assigned to containers as they join the network.

3. See similar info from common networking tools:

```
[centos@node-1 ~]$ ip addr
```

   Note the `bridge` network's gateway corresponds to the IP of the `docker0` device in this list. `docker0` is the linux bridge itself, while `bridge` is the name of the default Docker network that uses that bridge.

4. Use `brctl` to see connections to the `docker0` bridge:

```
[centos@node-1 ~]$ brctl show docker0

bridge name bridge id       STP enabled interfaces
docker0     8000.02427f12c30b   no
```

   At the moment, there are no connections to `docker0`.

11.2 Connecting Containers to `docker0`

1. Start a container and reexamine the network; the container is listed as connected to the network, with an IP assigned to it from the bridge network's subnet:

```
[centos@node-1 ~]$ docker container run --name u1 -dt centos:7
[centos@node-1 ~]$ docker network inspect bridge

...
    "Containers": {
        "11da9b7db065f971f78aebf14b706b0b85f07ec10dbf6f0773b1603f48697961": {
            "Name": "u1",
            "EndpointID": "670c4950816c43da255f44399d706fff3a7934831defce625f3ff8945000b1b0",
            "MacAddress": "02:42:ac:11:00:02",
            "IPv4Address": "172.17.0.2/16",
            "IPv6Address": ""
        }
    },
...
```

2. Inspect the network interfaces with `ip` and `brctl` again, now that you have a container running:

```
[centos@node-1 ~]$ ip addr

...
5: veth6f244c3@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state U
    link/ether aa:71:82:6c:f3:88 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::a871:82ff:fe6c:f388/64 scope link
       valid_lft forever preferred_lft forever

[centos@node-1 ~]$ brctl show docker0
```



```
bridge name bridge id       STP enabled interfaces
docker0     8000.02427f12c30b   no      veth6f244c3
```

`ip addr` indicates a veth endpoint has been created and plugged into the `docker0` bridge, as indicated by `master docker0`, and that it is connected to device index 4 in this case (indicated by the `@if4` suffix to the veth device name above). Similarly, `brctl` now shows this veth connection on `docker0` (notice that the ID for the veth connection matches in both utilities).

3. Launch a bash shell in your container, and look for the `eth0` device therein:

```
[centos@node-1 ~]$ docker container exec -it u1 bash
[root@11da9b7db065 /]# yum install -y iproute
[root@11da9b7db065 /]# ip addr

...
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
```

We see that the `eth0` device in this namespace is in fact the device that the veth connection in the host namespace indicated it was attached to, and vice versa - `eth0@if5` indicates it is plugged into networking interface number 5, which we saw above was the other end of the veth connection. Docker has created a veth connection with one end in the host's `docker0` bridge, and the other providing the `eth0` device in the container.

11.3 Defining Additional Bridge Networks

In the last step, we investigated the default bridge network; now let's try making our own. User defined bridge networks work exactly the same as the default one, but provide DNS lookup by container name, and are firewalled from other networks by default.

1. Create a bridge network by using the `bridge` driver with `docker network create`:

```
[centos@node-1 ~]$ docker network create --driver bridge my_bridge
```

2. Launch a container connected to your new network via the `--network` flag:

```
[centos@node-1 ~]$ docker container run --name=u2 --network=my_bridge -dt centos:7
```

3. Use the `inspect` command to investigate the network settings of this container:

```
[centos@node-1 ~]$ docker container inspect u2
```

`my_bridge` should be listed under the `Networks` key.

4. Launch another container, this time interactively:

```
[centos@node-1 ~]$ docker container run --name=u3 --network=my_bridge -it centos:7
```

5. From inside container u3, ping u2 by name: `ping u2`. The ping succeeds, since Docker is able to resolve container names when they are attached to a custom network.

6. Try starting a container on the default network, and pinging `u1` by name:

```
[centos@node-1 ~]$ docker container run centos:7 ping u1

ping: u1: Name or service not known
```

The ping fails; even though the containers are both attached to the `bridge` network, Docker does not provide name lookup on this default network. Try the same command again, but using `u1`'s IP instead of name, and you should be successful.

7. Finally, try pinging u1 by IP, this time from container u2:

```
[centos@node-1 ~]$ docker container exec u2 ping <u1 IP>
```

   The ping fails, since the containers reside on different networks; all Docker networks are firewalled from each other by default.

8. Clean up your containers and networks:

```
[centos@node-1 ~]$ docker container rm -f $(docker container ls -aq)
[centos@node-1 ~]$ docker network rm my_bridge
```

11.4 Conclusion

In this exercise, you explored the fundamentals of container networking. The key take away is that *containers on separate networks are firewalled from each other by default*. This should be leveraged as much as possible to harden your applications; if two containers don't need to talk to each other, put them on separate networks.

You also explored a number of API objects:

- `docker network ls` lists all networks on the host
- `docker network inspect <network name>` gives more detailed info about the named network
- `docker network create --driver <driver> <network name>` creates a new network using the specified driver; so far, we've only seen the `bridge` driver, for creating a linux bridge based network.
- `docker network connect <network name> <container name or id>` connects the specified container to the specified network after the container is running; the `--network` flag in `docker container run` achieves the same result at container launch.
- `docker container inspect <container name or id>` yields, among other things, information about the networks the specified container is connected to.

12 Container Port Mapping

By the end of this exercise, you should be able to:

- Forward traffic from a port on the docker host to a port inside a container's network namespace
- Define ports to automatically expose in a Dockerfile

12.1 Port Mapping at Runtime

1. Run an nginx container with no special port mappings:

```
[centos@node-1 ~]$ docker container run -d nginx
```

   nginx stands up a landing page at `<ip>:80`; try to visit this at your host or container's IP, and it won't be visible; no external traffic can make it past the linux bridge's firewall to the nginx container.

2. Now run an nginx container and map port 80 on the container to port 5000 on your host using the `-p` flag:

```
[centos@node-1 ~]$ docker container run -d -p 5000:80 nginx
```

   Note that the syntax is: `-p [host-port]:[container-port]`.

3. Verify the port mappings with the `docker container port` command

```
[centos@node-1 ~]$ docker container port <container id>

80/tcp -> 0.0.0.0:5000
```

4. Visit your nginx landing page at `<host ip>:5000`, e.g. using `curl -4 localhost:5000`, just to confirm it's working as expected.

12.2 Exposing Ports from the Dockerfile

1. In addition to manual port mapping, we can expose some ports in a Dockerfile for automatic port mapping on container startup. In a fresh directory `~/port`, create a Dockerfile:

```
FROM nginx

EXPOSE 80
```

2. Build your image as `my_nginx`:

```
[centos@node-1 port]$ docker image build -t my_nginx .
```

3. Use the `-P` flag when running to map all ports mentioned in the `EXPOSE` directive:

```
[centos@node-1 port]$ docker container run -d -P my_nginx
```

4. Use `docker container ls` or `docker container port` to find out what host ports were used, and visit your nginx landing page in a browser at `<node-1 public IP>:<port>`.

5. Clean up your containers:

```
[centos@node-1 port]$ docker container rm -f $(docker container ls -aq)
```

12.3 Conclusion

In this exercise, we saw how to explicitly map ports from our container's network stack onto ports of our host at runtime with the `-p` option to `docker container run`, or more flexibly in our Dockerfile with `EXPOSE`, which will result in the listed ports inside our container being mapped to random available ports on our host. In both cases, Docker is writing iptables rules to forward traffic from the host to the appropriate port in the container's network namespace.

13 Starting a Compose App

By the end of this exercise, you should be able to:

- Read a basic docker compose yaml file and understand what components it is declaring
- Start, stop, and inspect the logs of an application defined by a docker compose file

13.1 Inspecting a Compose App

1. Download the Dockercoins app from github:

```
[centos@node-0 ~]$ git clone -b ee2.1 \
    https://github.com/docker-training/orchestration-workshop.git
[centos@node-0 ~]$ cd orchestration-workshop/dockercoins
```

   This app consists of 5 services: a random number generator `rng`, a `hasher`, a backend `worker`, a `redis` queue, and a `web` frontend; the code you just downloaded has the source code for each process and a Dockerfile to containerize each of them.

2. Have a brief look at the source for each component of your application. Each folder under `~/orchestration-workshop/dockercoins` contains the application logic for the component, and a Dockerfile for building that logic into a Docker image. We've pre-built these images as `training/dockercoins-rng:1.0`, `training/dockercoins-worker:1.0` et cetera, so no need to build them yourself.

3. Have a look in `docker-compose.yml`; especially notice the `services` section. Each block here defines a different Docker service. They each have exactly one image which containers for this service will be started from, as well as other configuration details like network connections and port exposures. Full syntax for Docker Compose files can be found here: https://dockr.ly/2iHUpeX.

13.2 Starting the App

1. Stand up the app:

```
[centos@node-0 dockercoins]$ docker-compose up
```

   After a moment, your app should be running; visit <node 0 public IP>:8000 to see the web frontend visualizing your rate of Dockercoin mining.

2. Logs from all the running services are sent to STDOUT. Let's send this to the background instead; kill the app with CTRL+C, sending a SIGTERM to all running processes; some exit immediately, while others wait for a 10s timeout before being killed by a subsequent SIGKILL. Start the app again in the background:

```
[centos@node-0 dockercoins]$ docker-compose up -d
```

3. Check out which containers are running thanks to Compose:

```
[centos@node-0 dockercoins]$ docker-compose ps

        Name                       Command               State          Ports
-------------------------------------------------------------------------------------
dockercoins_hasher_1   ruby hasher.rb                   Up      0.0.0.0:8002->80/tcp
dockercoins_redis_1    docker-entrypoint.sh redis ...   Up      6379/tcp
dockercoins_rng_1      python rng.py                    Up      0.0.0.0:8001->80/tcp
dockercoins_webui_1    node webui.js                    Up      0.0.0.0:8000->80/tcp
dockercoins_worker_1   python worker.py                 Up
```

4. Compare this to the usual docker container ls; do you notice any differences? If not, start a couple of extra containers using docker container run..., and check again.

13.3 Viewing Logs

1. See logs from a Compose-managed app via:

```
[centos@node-0 dockercoins]$ docker-compose logs
```

2. The logging API in Compose follows the main Docker logging API closely. For example, try following the tail of the logs just like you would for regular container logs:

```
[centos@node-0 dockercoins]$ docker-compose logs --tail 10 --follow
```

   Note that when following a log, CTRL+S and CTRL+Q pauses and resumes live following; CTRL+C exits follow mode as usual.

13.4 Conclusion

In this exercise, you saw how to start a pre-defined Compose app, and how to inspect its logs. Application logic was defined in each of the five images we used to create containers for the app, but the manner in which those containers were created was defined in the docker-compose.yml file; all runtime configuration for each container is captured in this manifest. Finally, the different elements of Dockercoins communicated with each other via service name; the Docker daemon's internal DNS was able to resolve traffic destined for a service, into the IP or MAC address of the corresponding container.

14 Scaling a Compose App

By the end of this exercise, you should be able to:

- Scale a service from Docker Compose up or down

14.1 Scaling a Service

Any service defined in our `docker-compose.yml` can be scaled up from the Compose API; in this context, 'scaling' means launching multiple containers for the same service, which Docker Compose can route requests to and from.

1. Scale up the `worker` service in our Dockercoins app to have two workers generating coin candidates by redeploying the app with the `--scale` flag, while checking the list of running containers before and after:

```
[centos@node-0 dockercoins]$ docker-compose ps
[centos@node-0 dockercoins]$ docker-compose up -d --scale worker=2
[centos@node-0 dockercoins]$ docker-compose ps

        Name                       Command               State          Ports
-------------------------------------------------------------------------------------
dockercoins_hasher_1    ruby hasher.rb                   Up      0.0.0.0:8002->80/tcp
dockercoins_redis_1     docker-entrypoint.sh redis ...   Up      6379/tcp
dockercoins_rng_1       python rng.py                    Up      0.0.0.0:8001->80/tcp
dockercoins_webui_1     node webui.js                    Up      0.0.0.0:8000->80/tcp
dockercoins_worker_1    python worker.py                 Up
dockercoins_worker_2    python worker.py                 Up
```

   A new worker container has appeared in your list of containers.

2. Look at the performance graph provided by the web frontend; the coin mining rate should have doubled. Also check the logs using the logging API we learned in the last exercise; you should see a second `worker` instance reporting.

14.2 Investigating Bottlenecks

1. Try running `top` to inspect the system resource usage; it should still be fairly negligible. So, keep scaling up your workers:

```
[centos@node-0 dockercoins]$ docker-compose up -d --scale worker=10
[centos@node-0 dockercoins]$ docker-compose ps
```

2. Check your web frontend again; has going from 2 to 10 workers provided a 5x performance increase? It seems that something else is bottlenecking our application; any distributed application such as Dockercoins needs tooling to understand where the bottlenecks are, so that the application can be scaled intelligently.

3. Look in `docker-compose.yml` at the `rng` and `hasher` services; they're exposed on host ports 8001 and 8002, so we can use `httping` to probe their latency.

```
[centos@node-0 dockercoins]$ httping -c 5 localhost:8001
[centos@node-0 dockercoins]$ httping -c 5 localhost:8002
```

   `rng` on port 8001 has the much higher latency, suggesting that it might be our bottleneck. A random number generator based on entropy won't get any better by starting more instances on the same machine; we'll need a way to bring more nodes into our application to scale past this, which we'll explore in the next unit on Docker Swarm.

4. For now, shut your app down:

```
[centos@node-0 dockercoins]$ docker-compose down
```

14.3 Conclusion

In this exercise, we saw how to scale up a service defined in our Compose app using the `--scale` flag. Also, we saw how crucial it is to have detailed monitoring and tooling in a microservices-oriented application, in order to correctly identify bottlenecks and take advantage of the simplicity of scaling with Docker.

15 Creating a Swarm

By the end of this exercise, you should be able to:

- Create a swarm in high availability mode
- Set default address pools
- Check necessary connectivity between swarm nodes
- Configure the swarm's TLS certificate rotation

15.1 Starting Swarm

1. On `node-0`, initialize swarm and create a cluster with a default address pool for a discontiguous address range of 10.85.0.0/16 and 10.91.0.0/16 with a default subnet size of 128 addresses. This will be your first manager node:

```
[centos@node-0 ~]$ docker swarm init \
    --default-addr-pool 10.85.0.0/16 \
    --default-addr-pool 10.91.0.0/16 \
    --default-addr-pool-mask-length 25
```

2. Confirm that Swarm Mode is active and that the default address pool configuration has been registered by inspecting the output of:

```
[centos@node-0 ~]$ docker system info

...
Swarm: active
...
    Default Address Pool: 10.85.0.0/16  10.91.0.0/16
    SubnetSize: 25
...
```

3. See all nodes currently in your swarm by doing:

```
[centos@node-0 ~]$ docker node ls
```

   A single node is reported in the cluster.

4. Change the certificate rotation period from the default of 90 days to one week, and rotate the certificate now:

```
[centos@node-0 ~]$ docker swarm ca --rotate --cert-expiry 168h
```

   Note that the `docker swarm ca [options]` command *must* receive the `--rotate` flag, or all other flags will be ignored.

5. Display UDP and TCP activity on your manager:

```
[centos@node-0 ~]$ sudo netstat -plunt
```

   You should see (at least) TCP+UDP 7946, UDP 4789, and TCP 2377. What are each of these ports for?

15.2 Adding Workers to the Swarm

A single node swarm is not a particularly interesting swarm; let's add some workers to really see Swarm Mode in action.

1. On your manager node (`node-0`), get the swarm 'join token' you'll use to add worker nodes to your swarm:

```
[centos@node-0 ~]$ docker swarm join-token worker
```

2. SSH to `node-1`.
3. Paste in the join token you found in the first step above. `node-1` will join the swarm as a worker.

4. Inspect the network on `node-1` with `sudo netstat -plunt` like you did for the manager node. Are the same ports open? Why or why not?
5. Do `docker node ls` on the manager again, and you should see both your nodes and their status; note that `docker node ls` won't work on a worker node, as the cluster status is maintained only by the manager nodes.
6. Finally, use the same join token to add two more workers (`node-2` and `node-3`) to your swarm. When you're done, confirm that `docker node ls` on your one manager node reports 4 nodes in the cluster - one manager, and three workers.

15.3 Promoting Workers to Managers

At this point, our swarm has a single manager, `node-0`. If this node goes down, we'll lose the ability to maintain and schedule workloads on our swarm. In a real deployment, this is unacceptable; we need some redundancy to our system, and Swarm achieves this by allowing a raft consensus of multiple managers to preserve swarm state.

1. Promote two of your workers to manager status by executing, on the current manager node:

```
[centos@node-0 ~]$ docker node promote node-1 node-2
```

2. Finally, do a `docker node ls` to check and see that you now have three managers. Note that manager nodes also count as worker nodes - tasks can still be scheduled on them as normal.

15.4 Conclusion

In this exercise, you set up a basic high-availability swarm. In practice, it is crucial to have at least 3 (and always an odd number) of managers in order to ensure high availability of your cluster, and to ensure that the management, control, and data plane communications a swarm maintains can proceed unimpeded between all nodes.

16 Starting a Service

By the end of this exercise, you should be able to:

- Schedule a docker service across a swarm
- Predict and understand the scoping behavior of docker overlay networks
- Scale a service on swarm up or down
- Force swarm to spread workload out across user-defined divisions in a datacenter

16.1 Creating an Overlay Network and Service

1. Create a multi-host overlay network to connect your service to:

```
[centos@node-0 ~]$ docker network create --driver overlay my_overlay
```

2. Verify that the network subnet was taken from the address pool defined when creating your swarm:

```
[centos@node-0 ~]$ docker network inspect my_overlay

...
"Subnet": "10.85.0.0/25",
"Gateway": "10.85.0.1"
...
```

   The overlay network has been assigned a subnet from the address pool we specified when creating our swarm.

3. Create a service featuring an `alpine` container pinging Google resolvers, plugged into your overlay network:


```
[centos@node-0 ~]$ docker service create --name pinger \
    --network my_overlay alpine ping 8.8.8.8
```

Note the syntax is a lot like `docker container run`; an image (`alpine`) is specified, followed by the PID 1 process for that container (`ping 8.8.8.8`).

4. Get some information about the currently running services:

```
[centos@node-0 ~]$ docker service ls
```

5. Check which node the container was created on:

```
[centos@node-0 ~]$ docker service ps pinger
```

6. SSH into the node you found in the last step (call this `node-x`), find the container ID with `docker container ls`, and check its logs with `docker container logs <container ID>`. The results of the ongoing ping should be visible.

7. Inspect the `my_overlay` network on the node running your pinger container:

```
[centos@node-x ~]$ docker network inspect my_overlay
```

You should be able to see the container connected to this network, and a list of swarm nodes connected to this network under the `Peers` key. Also notice the correspondence between the container IPs and the subnet assigned to the network under the `IPAM` key - this is the subnet from which container IPs on this network are drawn.

8. Connect to your worker node, `node-3`, and list your networks:

```
[centos@node-3 ~]$ docker network ls
```

If the container for your service is **not** running here, you won't see the `my_overlay` network, since overlays only operate on nodes running containers attached to the overlay. On the other hand, if your container did get scheduled on `node-3`, you'll be able to see `my-overlay` as you should expect.

9. Connect to any manager node (`node-0`, `node-1` or `node-2`) and list the networks again. This time you will be able to see the network *whether or not* this manager has a container running on it for your `pinger` service; all managers maintain knowledge of all overlay networks.

10. On the same manager, inspect the `my_overlay` network again. If this manager does happen to have a container for the service scheduled on it, you'll be able to see the `Peers` list like above; if there is no container scheduled for the service on this node, the `Peers` list will be absent. `Peers` are maintained by Swarm's gossip control plane, which is scoped to only include nodes with running containers attached to the same overlay network.

16.2 Scaling a Service

1. Back on a manager node, scale up the number of concurrent tasks that our `alpine` service is running:

```
[centos@node-0 ~]$ docker service update pinger --replicas=8

pinger
overall progress: 8 out of 8 tasks
1/8: running   [==================================================>]
2/8: running   [==================================================>]
3/8: running   [==================================================>]
4/8: running   [==================================================>]
5/8: running   [==================================================>]
6/8: running   [==================================================>]
7/8: running   [==================================================>]
8/8: running   [==================================================>]
verify: Service converged
```


2. Now run `docker service ps pinger` to inspect the service. How were tasks distributed across your swarm?
3. Use `docker network inspect my_overlay` again on a node that has a `pinger` container running. More nodes appear connected to this network under the `Peers` key, since all these nodes started gossiping amongst themselves when they attached containers to the `my_overlay` network.

16.3 Inspecting Service Logs

1. In a previous step, you looked at the container logs for an individual task in your service; manager nodes can assemble all logs for all tasks of a given service by doing:

```
[centos@node-0 ~]$ docker service logs pinger
```

   The ping logs for all 8 pinging containers will be displayed.

2. If instead you'd like to see the logs of a single task, on a manager node run `docker service ps pinger`, choose any task ID, and run `docker service logs <task ID>`. The logs of the individual task are returned; compare this to what you did above to fetch the same information with `docker container logs`.

16.4 Scheduling Topology-Aware Services

By default, the Swarm scheduler will try to schedule an equal number of containers on all nodes, but in practice it is wise to consider datacenter segmentation; spreading tasks across datacenters or availability zones keeps the service available even when one such segment goes down.

1. Add a label `datacenter` with value `east` to two nodes of your swarm:

```
[centos@node-0 ~]$ docker node update --label-add datacenter=east node-0
[centos@node-0 ~]$ docker node update --label-add datacenter=east node-1
```

2. Add a label `datacenter` with value `west` to the other two nodes:

```
[centos@node-0 ~]$ docker node update --label-add datacenter=west node-2
[centos@node-0 ~]$ docker node update --label-add datacenter=west node-3
```

3. Create a service using the `--placement-pref` flag to spread across node labels:

```
[centos@node-0 ~]$ docker service create --name my_proxy \
    --replicas=2 --publish 8000:80 \
    --placement-pref spread=node.labels.datacenter \
    nginx
```

   There should be `nginx` containers present on nodes with every possible value of the `node.labels.datacenter` label, one in `datacenter=east` nodes, and one in `datacenter=west` nodes.

4. Use `docker service ps my_proxy` as above to check that replicas got spread across the datacenter labels.

16.5 Updating Service Configuration

1. If a container doesn't need to write to its filesystem, it should *always* be run in read-only mode, for security purposes. Update your service to use read-only containers:

```
[centos@node-0 ~]$ docker service update pinger --read-only

pinger
overall progress: 2 out of 8 tasks
1/8: running   [==================================================>]
2/8: running   [==================================================>]
3/8: ready     [=====================================>            ]
4/8:
```

```
5/8:
6/8:
7/8:
8/8:
```

Over the next few seconds, you should see tasks for the pinger service shutting down and restarting; this is the swarm manager replacing old containers which no longer match their desired state (using a read-only filesystem), with new containers that match the new configuration.

Once all containers for the pinger service have been restarted, try connecting to the container and creating a file to convince yourself this worked as expected.

16.6 Cleanup

1. Remove all existing services, in preparation for future exercises:

```
[centos@node-0 ~]$ docker service rm $(docker service ls -q)
```

16.7 Conclusion

In this exercise, we saw the basics of creating, scheduling and updating services. A common mistake people make is thinking that a service is just the containers scheduled by the service; in fact, a Docker service is the definition of *desired state* for those containers. Changing a service definition does not in general change containers directly; it causes them to get rescheduled by Swarm in order to match their new desired state.

17 Node Failure Recovery

By the end of this exercise, you should be able to:

- Anticipate swarm scheduling decisions when nodes fail and recover
- Force swarm to reallocate workload across a swarm

17.1 Setting up a Service

1. Set up a `myProxy` service with four replicas on one of your manager nodes:

```
[centos@node-0 ~]$ docker service create --replicas 4 --name myProxy nginx
```

2. Now watch the output of `docker service ps` on the same node:

```
[centos@node-0 ~]$ watch docker service ps myProxy
```

This should be stable for now, but will let us monitor scheduling updates as we interfere with the rest of our swarm.

17.2 Simulating Node Failure

1. SSH into `node-3`, and simulate a node failure by rebooting it:

```
[centos@node-3 ~]$ sudo reboot now
```

2. Back on your manager node, watch the updates to `docker service ps`; what happens to the task running on the rebooted node? Look at its desired state, any other tasks that get scheduled with the same name, and keep watching until `node-3` comes back online.

17.3 Force Rebalancing

By default, if a node fails and rejoins a swarm it *will not* get its old workload back; if we want to redistribute workload across a swarm after new nodes join (or old nodes rejoin), we need to force-rebalance our tasks

1. Back on the manager node, exit the watch mode with `CTRL+C`.
2. Force rebalance the tasks:

```
[centos@node-0 ~]$ docker service update --force myProxy
```

3. After the service converges, check which nodes the service tasks are scheduled on:

```
[centos@node-0 ~]$ docker service ps myProxy
... NAME            NODE                DESIRED STATE       CURRENT STATE
... myProxy.1       node-0              Running             Running about a minute ago
...  \_ myProxy.1   node-0              Shutdown            Shutdown about a minute ago
... myProxy.2       node-3              Running             Running about a minute ago
...  \_ myProxy.2   node-1              Shutdown            Shutdown about a minute ago
... myProxy.3       node-1              Running             Running about a minute ago
...  \_ myProxy.3   node-2              Shutdown            Shutdown about a minute ago
... myProxy.4       node-2              Running             Running about a minute ago
...  \_ myProxy.4   node-0              Shutdown            Shutdown about a minute ago
...  \_ myProxy.4   node-3              Shutdown            Shutdown 2 minutes ago
```

The _ shape indicate *ancestor* tasks which have been shut down and replaced by a new task, typically after reconfiguring the service or rebalancing like we've done here. Once the rebalance is complete, the current tasks for the `myProxy` service should be evenly distributed across your swarm.

17.4 Cleanup

1. On your manager node, remove all existing services, in preparation for future exercises:

```
[centos@node-0 ~]$ docker service rm $(docker service ls -q)
```

17.5 Conclusion

In this exercise, you saw swarm's scheduler in action - when a node is lost from the swarm, tasks are automatically rescheduled to restore the state of our services. Note that nodes joining or rejoining the swarm do not get workload automatically reallocated from existing nodes to them; rescheduling only happens when tasks crash, services are first scheduled, or you force a reschedule as above.

18 Routing Traffic to Docker Services

By the end of this exercise, you should be able to:

- Anticipate how swarm will load balance traffic across a service with more than one replica
- Publish a port on every swarm member that forwards all incoming traffic to the virtual IP of a swarm service

18.1 Observing Load Balancing

1. Start by deploying a simple service which spawns containers that echo back their hostname when `curl`'ed:

```
[centos@node-0 ~]$ docker service create --name who-am-I \
    --publish 8000:8000 \
    --replicas 3 training/whoami:latest
```


2. Run `curl -4 localhost:8000` and observe the output. You should see something similar to the following:

```
[centos@node-0 ~]$ curl -4 localhost:8000
I'm a7e5a21e6e26
```

   Take note of the response. In this example, our value is a7e5a21e6e26. The `whoami` containers uniquely identify themselves by returning their respective hostname. So each one of our `whoami` instances should have a different value.

3. Run `curl -4 localhost:8000` again. What can you observe? Notice how the value changes each time. This shows us that the routing mesh has sent our 2nd request over to a different container, since the value was different.

4. Repeat the command two more times. What can you observe? You should see one new value and then on the 4th request it should revert back to the value of the first container. In this example that value is a7e5a21e6e26.

5. Scale the number of tasks for our `who-am-I` service to 6:

```
[centos@node-0 ~]$ docker service update who-am-I --replicas=6
```

6. Now run `curl -4 localhost:8000` multiple times again. Use a loop like this:

```
[centos@node-0 ~]$ for n in {1..10}; do curl localhost:8000 -4; done

I'm 263fc24d0789
I'm 57ca6c0c0eb1
I'm c2ee8032c828
I'm c20c1412f4ff
I'm e6a88a30481a
I'm 86e262733b1e
I'm 263fc24d0789
I'm 57ca6c0c0eb1
I'm c2ee8032c828
I'm c20c1412f4ff
```

   You should be able to observe some new values. Note how the values repeat after the 6th curl command.

18.2 Using the Routing Mesh

1. Run an nginx service and expose the service port 80 on port 8080:

```
[centos@node-0 ~]$ docker service create --name nginx --publish 8080:80 nginx
```

2. Check which node your nginx service task is scheduled on:

```
[centos@node-0 ~]$ docker service ps nginx
```

3. Open a web browser and hit the IP address of that node at port 8080. You should see the NGINX welcome page. Try the same thing with the IP address of any other node in your cluster (using port 8080). No matter which swarm node IP you hit, the request gets forwarded to nginx by the routing mesh.

18.3 Cleanup

1. Remove all existing services, in preparation for future exercises:

```
[centos@node-0 ~]$ docker service rm $(docker service ls -q)
```

18.4 Conclusion

In these examples, you saw that requests to an exposed service will be automatically load balanced across all tasks providing that service. Furthermore, exposed services are reachable on all nodes in the swarm - whether they are

running a container for that service or not.

19 Dockercoins On Swarm

By the end of this exercise, you should be able to:

- Deploy an application on swarm as a 'stack', using a docker compose file
- Get some high-level monitoring information about the services and tasks running as part of a stack

19.1 Deploying a Stack

1. Deploy Dockercoins as a *stack* on your swarm, from `node-0`:

```
[centos@node-0 ~]$ cd ~/orchestration-workshop/dockercoins
[centos@node-0 dockercoins]$ docker stack deploy -c=docker-compose.yml dc
```

2. Check and see how your services are doing:

```
[centos@node-0 dockercoins]$ docker stack services dc

ID            NAME       MODE        REPLICAS  IMAGE                             PORTS
7tcaa3d3g9d2  dc_webui   replicated  1/1       training/dockercoins-webui:1.0    *:8000->80/tcp
mfq8i9cr4rcj  dc_rng     replicated  1/1       training/dockercoins-rng:1.0      *:8001->80/tcp
sbywya5yyrus  dc_redis   replicated  1/1       redis:latest
tk1ydqu794ng  dc_hasher  replicated  1/1       training/dockercoins-hasher:1.0   *:8002->80/tcp
v6835nwsek2n  dc_worker  replicated  1/1       training/dockercoins-worker:1.0
```

   Notice the `REPLICAS` column in the output of above command; this shows how many of your desired replicas are running. At first, a few might show 0/1; before those tasks can start, the worker nodes will need to download the appropriate images from Docker Hub.

3. Wait a minute or two, and try `docker stack services dc` again; once all services show 100% of their replicas are up, things are running properly and you can point your browser to port 8000 on one of the swarm nodes (does it matter which one?). You should see a graph of your dockercoin mining speed, around 3 hashes per second.

4. Finally, check out the details of the tasks running in your stack with `stack ps`:

```
[centos@node-0 dockercoins]$ docker stack ps dc
```

   This shows the details of each running task scheduled by services in your stack, similar to `service ps`, but for each service in the stack. Notice that these containers have been scheduled across our swarm, not just on one node like Docker Compose did.

19.2 Conclusion

In this exercise, we stood up our first *stack*. A stack is a collection of docker components (services, networks, volumes, etc) that make up a full application, and we can create one directly from the same docker compose file we used to start an application on a single host using docker compose. A stack, however, will schedule workloads using Swarm, distributed across our cluster.

20 Scaling and Scheduling Services

By the end of this exercise, you should be able to:

- Define the desired number of containers running as part of a service via the `deploy` block in a docker compose file

- Schedule services in replicated mode to ensure exactly one replica runs on every node in your swarm

20.1 Scaling up a Service

If we've written our services to be stateless, we might hope for linear performance scaling in the number of replicas of that service. For example, our `worker` service requests a random number from the `rng` service and hands it off to the `hasher` service; the faster we make those requests, the higher our throughput of dockercoins should be, as long as there are no other confounding bottlenecks.

1. Modify the `worker` service definition in `docker-compose.yml` to set the number of replicas to create using the `deploy` and `replicas` keys:

```
worker:
  image: training/dockercoins-worker:1.0
  networks:
  - dockercoins
  deploy:
     replicas: 2
```

2. Update your app by running the same command you used to launch it in the first place, and check to see when your new worker replica is up and running:

```
[centos@node-0 dockercoins]$ docker stack deploy -c docker-compose.yml dc
[centos@node-0 dockercoins]$ docker service ps dc_worker
```

3. Once both replicas of the `worker` service are live, check the web frontend; you should see about double the number of hashes per second, as expected.
4. Scale up even more by changing the `worker` replicas to 10. A small improvement should be visible, but certainly not an additional factor of 5. Something else is bottlenecking dockercoins.

20.2 Scheduling Services

Something other than `worker` is bottlenecking dockercoins's performance; the first place to look is in the services that `worker` directly interacts with.

1. The `rng` and `hasher` services are exposed on host ports 8001 and 8002, so we can use `httping` to probe their latency:

```
[centos@node-0 dockercoins]$ httping -c 5 localhost:8001
[centos@node-0 dockercoins]$ httping -c 5 localhost:8002
```

   `rng` is much slower to respond, suggesting that it might be the bottleneck. If this random number generator is based on an entropy collector (random voltage microfluctuations in the machine's power supply, for example), it won't be able to generate random numbers beyond a physically limited rate; we need more machines collecting more entropy in order to scale this up. This is a case where it makes sense to run exactly one copy of this service per machine, via `global` scheduling (as opposed to potentially many copies on one machine, or whatever the scheduler decides as in the default `replicated` scheduling).

2. Modify the definition of our `rng` service in `docker-compose.yml` to be globally scheduled:

```
rng:
  image: training/dockercoins-rng:1.0
  networks:
  - dockercoins
  ports:
  - "8001:80"
  deploy:
    mode: global
```


3. Scheduling can't be changed on the fly, so we need to stop our app and restart it:

```
[centos@node-0 dockercoins]$ docker stack rm dc
[centos@node-0 dockercoins]$ docker stack deploy -c=docker-compose.yml dc
```

4. Check the web frontend again; the overall factor of 10 improvement (from ~3 to ~35 hashes per second) should now be visible.

20.3 Conclusion

In this exercise, you explored the performance gains a distributed application can enjoy by scaling a key service up to have more replicas, and by correctly scheduling a service that needs to be replicated across different hosts.

21 Updating a Service

By the end of this exercise, you should be able to:

- Update a swarm service's underlying image, controlling update parallelism, speed, and rollback contingencies

21.1 Creating Rolling Updates

1. First, let's change one of our services a bit: open `orchestration-workshop/dockercoins/worker/worker.py` in your favorite text editor, and find the following section:

```
def work_once():
    log.debug("Doing one unit of work")
    time.sleep(0.1)
```

   Change the `0.1` to a `0.01`. Save the file, exit the text editor.

2. Rebuild the worker image with a tag of `<Docker ID>/dockercoins-worker:1.1`, and push it to Docker Hub.

3. Start the update, and wait for it to converge:

```
[centos@node-0 ~]$ docker service update dc_worker \
    --image <Docker ID>/dockercoins-worker:1.1
```

   Tasks are updated to our new 1.1 image one at a time.

21.2 Parallelizing Updates

1. We can also set our updates to run in batches by configuring some options associated with each service. Change the update parallelism to 2 and the delay to 5 seconds on the `worker` service by editing its definition in the `docker-compose.yml`:

```
worker:
  image: training/dockercoins-worker:1.0
  networks:
  - dockercoins
  deploy:
    replicas: 10
    update_config:
      parallelism: 2
      delay: 5s
```

2. Roll back the worker service to 1.0:


```
[centos@node-0 ~]$ docker stack deploy -c=docker-compose.yml dc
```

3. On node-1, watch your updates:

```
[centos@node-1 ~]$ watch -n1 "docker service ps dc_worker \
    | grep -v Shutdown.*Shutdown"
```

You should see two tasks get shutdown and restarted with the 1.0 image every five seconds.

21.3 Auto-Rollback Failed Updates

In the event of an application or container failure on deployment, we'd like to automatically roll the update back to the previous version.

1. Update the worker service with some parameters to define rollback:

```
[centos@node-0 ~]$ docker service update \
    --update-failure-action=rollback \
    --update-max-failure-ratio=0.2 \
    --update-monitor=20s \
    dc_worker
```

These parameters will trigger a rollback if more than 20% of services tasks fail in the first 20 seconds after an update.

2. Make a broken version of the worker service to trigger a rollback with; try removing all the import commands at the top of worker.py, for example. Then rebuild the worker image with a tag <Docker ID>/dockercoins-worker:bugged, push it to Docker Hub, and attempt to update your service:

```
[centos@node-0 ~]$ docker image build -t <Docker ID>/dockercoins-worker:bugged .
[centos@node-0 ~]$ docker image push <Docker ID>/dockercoins-worker:bugged
[centos@node-0 ~]$ docker service update \
    dc_worker --image <Docker ID>/dockercoins-worker:bugged
```

3. The connection to node-1 running watch should show the :bugged tag getting deployed, failing, and rolling back to :1.0 automatically over the next minute or two.

21.4 Shutting Down a Stack

1. To shut down a running stack:

```
[centos@node-0 ~]$ docker stack rm <stack name>
```

Where the stack name can be found in the output of docker stack ls.

21.5 Conclusion

In this exercise, we explored deploying and redeploying an application as stacks and services. Note that relaunching a running stack updates all the objects it manages in the most non-disruptive way possible; there is usually no need to remove a stack before updating it. In production, rollback contingencies should always be used to cautiously upgrade images, cutting off potential damage before an entire service is taken down.

22 Installing Kubernetes

By the end of this exercise, you should be able to:

- Set up a Kubernetes cluster with one master and two nodes

22.1 Initializing Kubernetes

1. On node-0, initialize the cluster with kubeadm:

```
[centos@node-0 ~]$ sudo kubeadm init --pod-network-cidr=192.168.0.0/16 \
    --ignore-preflight-errors=SystemVerification
```

   If successful, the output will end with a join command:

```
...
You can now join any number of machines by running the following on each node
as root:

  kubeadm join 10.10.29.54:6443 --token wdytg5.q1w1f4dau7u6wk11 --discovery-token-ca-cert-hash sha2
```

2. To start using you cluster, you need to run:

```
[centos@node-0 ~]$ mkdir -p $HOME/.kube
[centos@node-0 ~]$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[centos@node-0 ~]$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

3. List all your nodes in the cluster:

```
[centos@node-0 ~]$ kubectl get nodes
```

   Which should output something like:

```
NAME      STATUS     ROLES     AGE       VERSION
node-0    NotReady   master    2h        v1.11.1
```

   The NotReady status indicates that we must install a network for our cluster.

4. Let's install the Calico network driver:

```
[centos@node-0 ~]$ kubectl apply -f https://bit.ly/2v9yaaV
```

5. After a moment, if we list nodes again, ours should be ready:

```
[centos@node-0 ~]$ kubectl get nodes -w
NAME      STATUS     ROLES     AGE       VERSION
node-0    NotReady   master    1m        v1.11.1
node-0    NotReady   master    1m        v1.11.1
node-0    NotReady   master    1m        v1.11.1
node-0    Ready     master    2m        v1.11.1
node-0    Ready     master    2m        v1.11.1
```

6. Execute the join command you found above when initializing Kubernetes on node-1 and node-2 (you'll need to add sudo to the start, and --ignore-preflight-errors=SystemVerification to the end), and then check the status back on node-0:

```
[centos@node-1 ~]$ sudo kubeadm join ... --ignore-preflight-errors=SystemVerification
[centos@node-2 ~]$ sudo kubeadm join ... --ignore-preflight-errors=SystemVerification
[centos@node-0 ~]$ kubectl get nodes
```

   After a few moments, there should be three nodes listed - all with the Ready status.

7. Let's see what system pods are running on our cluster:

```
[centos@node-0 ~]$ kubectl get pods -n kube-system
```

   which results in something similar to this:

```
NAME                                  READY     STATUS    RESTARTS   AGE
calico-etcd-pfhj4                     1/1       Running   1          5h
calico-kube-controllers-559c657d6d-ztk8c   1/1       Running   1          5h
calico-node-89k9v                     2/2       Running   0          4h
```



```
calico-node-brqxz                  2/2       Running   2          5h
calico-node-zsmh2                  2/2       Running   1          41s
coredns-78fcdf6894-gtj87           1/1       Running   1          5h
coredns-78fcdf6894-nz2kw           1/1       Running   1          5h
etcd-node-0                        1/1       Running   1          5h
kube-apiserver-node-0              1/1       Running   1          5h
kube-controller-manager-node-0     1/1       Running   1          5h
kube-proxy-qxfzt                   1/1       Running   0          41s
kube-proxy-vgrtm                   1/1       Running   0          4h
kube-proxy-ws2z5                   1/1       Running   0          5h
kube-scheduler-node-0              1/1       Running   1          5h
```

We can see the pods running on the master: etcd, api-server, controller manager and scheduler, as well as calico and DNS infrastructure pods deployed when we installed calico.

22.2 Conclusion

At this point, we have a Kubernetes cluster with one master and two workers ready to accept workloads.

23 Kubernetes Orchestration

By the end of this exercise, you should be able to:

- Define and launch basic pods, replicaSets and deployments using `kubectl`
- Get metadata, configuration and state information about a kubernetes object using `kubectl describe`
- Update an image for a pod in a running kubernetes deployment

23.1 Creating Pods

1. On your master node, create a yaml file `pod.yaml` to describe a simple pod with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
```

2. Deploy your pod:

```
[centos@node-0 ~]$ kubectl create -f pod.yaml
```

3. Confirm your pod is running:

```
[centos@node-0 ~]$ kubectl get pod demo
```

4. Get some metadata about your pod:

```
[centos@node-0 ~]$ kubectl describe pod demo
```

5. Delete your pod:

```
[centos@node-0 ~]$ kubectl delete pod demo
```

6. Modify `pod.yaml` to create a second container inside your pod:


```
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
  - name: sidecar
    image: centos:7
    command: ["ping"]
    args: ["8.8.8.8"]
```

7. Deploy this new pod, and create a bash shell inside the container named `sidecar`:

```
[centos@node-0 ~]$ kubectl create -f pod.yaml
[centos@node-0 ~]$ kubectl exec -c=sidecar -it demo -- /bin/bash
```

8. From within the `sidecar` container, fetch the nginx landing page on the default port 80 using `localhost`:

```
[root@demo /]# curl localhost:80
```

   You should see the html of the nginx landing page. Note **these containers can reach each other on localhost**, meaning they are sharing a network namespace. Now list the processes in your `sidecar` container:

```
[root@demo /]# ps -aux
```

   You should see the `ping` process we containerized, the shell we created to explore this container using `kubectl exec`, and the `ps` process itself - but no `nginx`. While a network namespace is shared between the containers, they still have their own PID namespace (for example).

9. Finally, remember to exit out of this pod, and delete it:

```
[root@demo /]# exit
[centos@node-0 ~]$ kubectl delete pod demo
```

23.2 Creating ReplicaSets

1. On your master `node-0`, create a yaml file `replicaset.yaml` to describe a simple replicaSet with the following content:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-demo
spec:
  replicas: 3
  selector:
    matchLabels:
      component: reverse-proxy
  template:
    metadata:
      labels:
        component: reverse-proxy
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
```


Notice especially the `replicas` key, which defines how many copies of this pod to create, and the `template` section; this defines the pod to replicate, and is described almost exactly like the first pod definition we created above. The difference here is the required presence of the `labels` key in the pod's metadata, which must match the `selector -> matchLabels` item in the specification of the replicaSet.

2. Deploy your replicaSet, and get some state information about it:

```
[centos@node-0 ~]$ kubectl create -f replicaset.yaml
[centos@node-0 ~]$ kubectl describe replicaset rs-demo
```

After a few moments, you should see something like

```
Name:         rs-demo
Namespace:    default
Selector:     component=reverse-proxy
Labels:       component=reverse-proxy
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  component=reverse-proxy
  Containers:
   nginx:
    Image:        nginx:1.7.9
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  35s   replicaset-controller  Created pod: rs-demo-jxmjj
  Normal  SuccessfulCreate  35s   replicaset-controller  Created pod: rs-demo-dmdtf
  Normal  SuccessfulCreate  35s   replicaset-controller  Created pod: rs-demo-j62fx
```

Note the replicaSet has created three pods as requested, and will reschedule them if they exit.

3. Try killing off one of your pods, and reexamining the output of the above `describe` command. The `<pod name>` comes from the last three lines in the output above, such as `rs-demo-jxmjj`:

```
[centos@node-0 ~]$ kubectl delete pod <pod name>
[centos@node-0 ~]$ kubectl describe replicaset rs-demo
```

The dead pod gets rescheduled by the replicaSet, similar to a failed task in Docker Swarm.

4. Delete your replicaSet:

```
[centos@node-0 ~]$ kubectl delete replicaset rs-demo
```

23.3 Creating Deployments

1. On your master `node-0`, create a yaml file `deployment.yaml` to describe a simple deployment with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
```



```
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
```

Notice this is the exact same structure as your replicaSet yaml above, but this time the `kind` is `Deployment`. Deployments create a replicaSet of pods, but add some deployment management functionality on top of them, such as rolling updates and rollback.

2. Spin up your deployment, and get some state information:

```
[centos@node-0 ~]$ kubectl create -f deployment.yaml
[centos@node-0 ~]$ kubectl describe deployment nginx-deployment
```

The `describe` command should return something like:

```
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Thu, 24 May 2018 04:29:18 +0000
Labels:                 <none>
Annotations:            deployment.kubernetes.io/revision=1
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.7.9
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-85f7784776 (3/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  10s   deployment-controller  Scaled up replica set nginx-deployment-85
```

Note the very last line, indicating this deployment actually created a replicaSet which it used to scale up to three pods.

3. List your replicaSets and pods:

```
[centos@node-0 ~]$ kubectl get replicaSet

NAME                           DESIRED   CURRENT   READY   AGE
nginx-deployment-79d4c74645   3         3         3       17s

[centos@node-0 ~]$ kubectl get pod

NAME                                 READY   STATUS    RESTARTS   AGE
nginx-deployment-79d4c74645-9mtzm   1/1     Running   0          22s
nginx-deployment-79d4c74645-k7wml   1/1     Running   0          22s
nginx-deployment-79d4c74645-rrfrf   1/1     Running   0          22s
```

   You should see one replicaSet and three pods created by your deployment, similar to the above.

4. Upgrade the nginx image from `1.7.9` to `1.9.1`:

```
[centos@node-0 ~]$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
```

5. After a few seconds, `kubectl describe` your deployment as above again. You should see that the image has been updated, and that the old replicaSet has been scaled down to 0 replicas, while a new replicaSet (with your updated image) has been scaled up to 3 pods. List your replicaSets one more time:

```
[centos@node-0 ~]$ kubectl get replicaSets
```

   You should see something like

```
NAME                           DESIRED   CURRENT   READY     AGE
nginx-deployment-69df9ccbf8   3         3         3         4m
nginx-deployment-85f7784776   0         0         0         9m
```

   Do a `kubectl describe replicaSet <replicaSet scaled down to 0>`; you should see that while no pods are running for this replicaSet, the old replicaSet's definition is still around so we can easily roll back to this version of the app if we need to.

6. Clean up your cluster:

```
[centos@node-0 ~]$ kubectl delete deployment nginx-deployment
```

23.4 Conclusion

In this exercise, you explored the basic scheduling objects of pods, replicaSets, and deployments. Each object is responsible for a different part of the orchestration stack; pods are the basic unit of scheduling, replicaSets do keep-alive and scaling, and deployments provide update and rollback functionality. In a sense, these objects all 'nest' one inside the next; by creating a deployment, you implicitly created a replicaSet which in turn created the corresponding pods. In most cases, you're better off creating deployments rather than replicaSets or pods directly; this way, you get all the orchestrating scheduling features you would expect in analogy to a Docker Swarm service.

24 Kubernetes Networking

By the end of this exercise, you should be able to:

- Predict what routing tables rules calico will write to each host in your cluster
- Route and load balance traffic to deployments using clusterIP and nodePort services
- Reconfigure a deployment into a daemonSet (analogous to changing scheduling from 'replicated' to 'global' in a swarm service)

24.1 Routing Traffic with Calico

1. Make sure you're on the master node `node-0`, and redeploy the nginx deployment defined in `deployment.yaml` from the last exercise.
2. List your pods:

```
[centos@node-0 ~]$ kubectl get pods
```

3. Get some metadata on one of the pods found in the last step:

```
[centos@node-0 ~]$ kubectl describe pods <pod name>
```

which in my case results in:

```
Name:               nginx-deployment-69df458bc5-bb87w
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               node-2/10.10.43.25
Start Time:         Thu, 09 Aug 2018 17:29:52 +0000
Labels:             app=nginx
                    pod-template-hash=2589014671
Annotations:        <none>
Status:             Running
IP:                 192.168.247.10
Controlled By:      ReplicaSet/nginx-deployment-69df458bc5
Containers:
  nginx:
    Container ID:   docker://26e8eac8d5a89b7cf2f2af762de88d7f4fa234174881626a1427b813c06b1362
    Image:          nginx:1.7.9
    Image ID:       docker-pullable://nginx@sha256:e3456c851a152494c3e4ff5fcc26f240206abac0c9d794af
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Thu, 09 Aug 2018 17:29:53 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fkf5d (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-fkf5d:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-fkf5d
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
```



```
  Normal  Scheduled  20s   default-scheduler  Successfully assigned default/nginx-deployment-69df45
  Normal  Pulled     19s   kubelet, node-2    Container image "nginx:1.7.9" already present on mach
  Normal  Created    19s   kubelet, node-2    Created container
  Normal  Started    19s   kubelet, node-2    Started container
[centos@node-0 ~]$ kubectl describe pods nginx-deployment-69df458bc5-bb87w
Name:               nginx-deployment-69df458bc5-bb87w
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               node-2/10.10.43.25
Start Time:         Thu, 09 Aug 2018 17:29:52 +0000
Labels:             app=nginx
                    pod-template-hash=2589014671
Annotations:        <none>
Status:             Running
IP:                 192.168.247.10
Controlled By:      ReplicaSet/nginx-deployment-69df458bc5
Containers:
  nginx:
    Container ID:   docker://26e8eac8d5a89b7cf2f2af762de88d7f4fa234174881626a1427b813c06b1362
    Image:          nginx:1.7.9
    Image ID:       docker-pullable://nginx@sha256:e3456c851a152494c3e4ff5fcc26f240206abac0c9d794af
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Thu, 09 Aug 2018 17:29:53 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fkf5d (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-fkf5d:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-fkf5d
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  1m    default-scheduler  Successfully assigned default/nginx-deployment-69df45
  Normal  Pulled     1m    kubelet, node-2    Container image "nginx:1.7.9" already present on mach
  Normal  Created    1m    kubelet, node-2    Created container
  Normal  Started    1m    kubelet, node-2    Started container
```

We can see that in our case the pod has been deployed to `node-2` as indicated near the top of the output, and the pod has an IP of `192.168.247.10`.

4. Have a look at the routing table on node-0 using `ip route`, which for my example looks like:

```
[centos@node-0 ~]$ ip route

default via 10.10.0.1 dev eth0
10.10.0.0/20 dev eth0  proto kernel  scope link  src 10.10.7.20
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
blackhole 192.168.39.192/26  proto bird
192.168.39.193 dev cali12b0eb5c038  scope link
192.168.39.194 dev calibe752d56965  scope link
192.168.84.128/26 via 10.10.24.89 dev tunl0  proto bird onlink
192.168.247.0/26 via 10.10.43.25 dev tunl0  proto bird onlink
```

   Notice the last line; this rule was written by Calico to send any traffic on the 192.168.247.0/26 subnet (which the pod we examined above is on) to the host at IP 10.10.43.25 via IP in IP as indicated by the `dev tunl0` entry. Look at your own routing table and list of VM IPs; what are the corresponding subnets, pod IPs and host IPs in your case? Does that make sense based on the host you found for the nginx pod above?

5. Curl your pod's IP on port 80 from node-0; you should see the HTML for the nginx landing page. By default this pod is reachable at this IP from anywhere in the Kubernetes cluster.

6. Head over to the node this pod got scheduled on (node-2 in the example above), and have a look at that host's routing table in the same way:

```
[centos@node-2 ~]$ ip route

default via 10.10.32.1 dev eth0
10.10.32.0/20 dev eth0  proto kernel  scope link  src 10.10.43.25
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
192.168.39.192/26 via 10.10.7.20 dev tunl0  proto bird onlink
192.168.84.128/26 via 10.10.24.89 dev tunl0  proto bird onlink
blackhole 192.168.247.0/26  proto bird
192.168.247.10 dev calia5daa4e7a1d  scope link
192.168.247.11 dev cali9ff153fb143  scope link
```

   Again notice the second-to-last line; this time, the pod IP is routed to a `cali***` device, which is a virtual ethernet endpoint in the host's network namespace, providing a point of ingress into that pod. Once again try `curl <pod IP>:80` - you'll see the nginx landing page html as before.

7. Back on node-0, fetch the logs generated by the pod you've been curling:

```
[centos@node-0 ~]$ kubectl logs <pod name>
10.10.52.135 - - [09/May/2018:13:58:42 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
192.168.84.128 - - [09/May/2018:14:00:41 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.29.0" "-"
```

   We see records of the curls we preformed above; like Docker containers, these logs are the STDOUT and STDERR of the containerized processes.

24.2 Routing and Load Balancing with Services

1. Above we were able to hit nginx at the pod IP, but there is no guarantee this pod won't get rescheduled to a new IP. If we want a stable IP for this deployment, we need to create a `ClusterIP` service. In a file `cluster.yaml` on your master node-0:

```
apiVersion: v1
kind: Service
metadata:
  name: cluster-demo
spec:
  selector:
```



```
    app: nginx
  ports:
  - port: 8080
    targetPort: 80
```

Create this service with `kubectl create -f cluster.yaml`. This maps the pod internal port 80 to the cluster wide external port 8080; furthermore, this IP and port will only be reachable from *within* the cluster. Also note the `selector: app: nginx` specification; that indicates that this service will route traffic to every pod that has `nginx` as the value of the `app` label in this namespace.

2. Let's see what services we have now:

```
[centos@node-0 ~]$ kubectl get services
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP    33m
cluster-demo   ClusterIP   10.104.201.93   <none>        8080/TCP   48s
```

The second one is the one we just created and we can see that a stable IP address and port `10.104.201.93:8080` has been assigned to our `nginx` service.

3. Let's try to access Nginx now, from any node in our cluster:

```
[centos@node-0 ~]$ curl <nginx CLUSTER-IP>:8080
```

which should return the nginx welcome page. Even if pods get rescheduled to new IPs, this clusterIP service will preserve a stable entrypoint for traffic to be load balanced across all pods matching the service's label selector.

4. ClusterIP services are reachable only from within the Kubernetes cluster. If you want to route traffic to your pods from an external network, you'll need a NodePort service. On your master `node-0`, create a file `nodeport.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-demo
spec:
  type: NodePort
  selector:
      app: nginx
  ports:
  - port: 8080
    targetPort: 80
```

And create this service with `kubectl create -f nodeport.yaml`. Notice this is exactly the same as the ClusterIP service definition, but now we're requesting a type NodePort.

5. Inspect this service's metadata:

```
[centos@node-0 ~]$ kubectl describe service nodeport-demo
```

Notice the NodePort field: this is a randomly selected port from the range 30000-32767 where your pods will be reachable externally. Try visiting your nginx deployment at any public IP of your cluster, and the port you found above, and confirming you can see the nginx landing page.

6. Clean up the objects you created in this section:

```
[centos@node-0 ~]$ kubectl delete deployment nginx-deployment
[centos@node-0 ~]$ kubectl delete service cluster-demo
[centos@node-0 ~]$ kubectl delete service nodeport-demo
```


24.3 Optional: Deploying DockerCoins onto the Kubernetes Cluster

1. First deploy Redis via `kubectl create deployment`:

```
[centos@node-0 ~]$ kubectl create deployment redis --image=redis
```

2. And now all the other deployments. To avoid too much typing we do that in a loop:

```
[centos@node-0 ~]$ for DEPLOYMENT in hasher rng webui worker; do
    kubectl create deployment $DEPLOYMENT --image=training/dockercoins-${DEPLOYMENT}:1.0
done
```

3. Let's see what we have:

```
[centos@node-0 ~]$ kubectl get pods -o wide -w
```

in my case the result is:

```
hasher-6c64f78655-rgjk5   1/1       Running   0          53s       10.36.0.1   node-2
redis-75586d7d7c-mmjg7    1/1       Running   0          5m        10.44.0.2   node-1
rng-d94d56d4f-twlwz       1/1       Running   0          53s       10.44.0.1   node-1
webui-6d8668984d-sqtt8    1/1       Running   0          52s       10.36.0.2   node-2
worker-56756ddbb8-lbv9r   1/1       Running   0          52s       10.44.0.3   node-1
```

pods have been distributed across our cluster.

4. We can also look at some logs:

```
[centos@node-0 ~]$ kubectl logs deploy/rng
[centos@node-0 ~]$ kubectl logs deploy/worker
```

The `rng` service (and also the `hasher` and `webui` services) seem to work fine but the `worker` service reports errors. The reason is that unlike on Swarm, Kubernetes does not automatically provide a stable networking endpoint for deployments. We need to create at least a `ClusterIP` service for each of our deployments so they can communicate.

5. List your current services:

```
[centos@node-0 ~]$ kubectl get services
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   46m
```

6. Expose the `redis`, `rng` and `hasher` internally to your cluster, specifying the correct internal port:

```
[centos@node-0 ~]$ kubectl expose deployment redis --port 6379
[centos@node-0 ~]$ kubectl expose deployment rng --port 80
[centos@node-0 ~]$ kubectl expose deployment hasher --port 80
```

7. List your services again:

```
[centos@node-0 ~]$ kubectl get services
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
hasher       ClusterIP   10.108.207.22    <none>        80/TCP     20s
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    47m
redis        ClusterIP   10.100.14.121    <none>        6379/TCP   31s
rng          ClusterIP   10.111.235.252   <none>        80/TCP     26s
```

Evidently `kubectl expose` creates `ClusterIP` services allowing stable, internal reachability for your deployments, much like you did via yaml manifests for your nginx deployment in the last section. See the `kubectl` api docs for more command-line alternatives to yaml manifests.

8. Get the logs of the worker again:

```
[centos@node-0 ~]$ kubectl logs deploy/worker
```


This time you should see that the worker recovered (give it at least 10 seconds to do so). The worker can now access the other services.

9. Now let's expose the webui to the public using a service of type NodePort:

```
[centos@node-0 ~]$ kubectl expose deploy/webui --type=NodePort --port 80
```

10. List your services one more time:

```
[centos@node-0 ~]$ kubectl get services
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
hasher       ClusterIP   10.108.207.22    <none>        80/TCP         2m
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP        49m
redis        ClusterIP   10.100.14.121    <none>        6379/TCP       2m
rng          ClusterIP   10.111.235.252   <none>        80/TCP         2m
webui        NodePort    10.108.88.182    <none>        80:32015/TCP   33s
```

Notice the NodePort service created for webui. This type of service provides similar behavior to the Swarm L4 mesh net: a port (32015 in my case) has been reserved across the cluster; any external traffic hitting any cluster IP on that port will be directed to port 80 inside a webui pod.

11. Visit your Dockercoins web ui at http://<node IP>:<port>, where <node IP> is the public IP address any of your cluster members. You should see the dashboard of our DockerCoins application.

12. Let's scale up the worker a bit and see the effect of it:

```
[centos@node-0 ~]$ kubectl scale deploy/worker --replicas=10
```

Observe the result of this scaling in the browser. We do not really get a 10-fold increase in throughput, just as when we deployed DockerCoins on swarm; the rng service is causing a bottleneck.

13. To scale up, we want to run an instance of rng on each node of the cluster. For this we use a DaemonSet. We do this by using a yaml file that captures the desired configuration, rather than through the CLI.

Create a file deploy-rng.yaml as follows:

```
[centos@node-0 ~]$ kubectl get deploy/rng -o yaml --export > deploy-rng.yaml
```

Note: --export will remove "cluster-specific" information

14. Edit this file to make it describe a DaemonSet instead of a Deployment:

- change kind to DaemonSet
- remove the replicas field
- remove the strategy block (which defines the rollout mechanism for a deployment)
- remove the status: {} line at the end

15. Now apply this YAML file to create the DaemonSet:

```
[centos@node-0 ~]$ kubectl apply -f deploy-rng.yaml
```

16. We can now look at the DaemonSet that was created:

```
[centos@node-0 ~]$ kubectl get daemonset

NAME      DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
rng       2         2         2         2            2           <none>          1m
```

Dockercoins performance should now improve, as illustrated by your web ui.

17. If we do a kubectl get all we will see that we now have both a deployment.apps/rng AND a daemonset.apps/rng. Deployments are not just converted to Daemon sets. Let's delete the rng deployment:

```
[centos@node-0 ~]$ kubectl delete deploy/rng
```

18. Clean up your resources when done:


```
[centos@node-0 ~]$ for D in redis hasher rng webui; \
    do kubectl delete svc/$D; done
[centos@node-0 ~]$ for D in redis hasher webui worker; \
    do kubectl delete deploy/$D; done
[centos@node-0 ~]$ kubectl delete ds/rng
```

19. Make sure that everything is cleared:

    ```
    [centos@node-0 ~]$ kubectl get all
    ```

    should only show the `svc/kubernetes` resource.

24.4 Conclusion

In this exercise, we looked at some of the key Kubernetes service objects that provide routing and load balancing for collections of pods; clusterIP for internal communication, analogous to Swarm's VIPs, and NodePort, for routing external traffic to an app similarly to Swarm's L4 mesh net. We also briefly touched on the inner workings of Calico, one of many Kubernetes network plugins and the one that ships natively with Docker's Enterprise Edition product. The key networking difference between Swarm and Kubernetes is their approach to default firewalling; while Swarm firewalls software defined networks automatically, all pods can reach all other pods on a Kube cluster, in Calico's case via the BGP-updated control plane and IP-in-IP data plane you explored above.

25 Orchestrating Secrets

By the end of this exercise, you should be able to:

- Declare secrets in Swarm and Kubernetes
- Provision secrets to a swarm service or kubernetes deployment
- Configure environment variables and application logic to consume secrets in either orchestrator

25.1 Prerequisites

- A Swarm with at least one node (`docker swarm init` on any node with Docker installed will do if you don't already have a swarm running).
- A Kubernetes cluster with at least one master and one worker (see the **Kubernetes Basics** demo in this book for setup instructions).

25.2 Creating Secrets

1. Create a new secret named `my-secret` with the value `abc1234` by using the following command to pipe STDIN to the secret value:

   ```
   [centos@node-0 ~]$ echo 'abc1234' | docker secret create my-secret -
   ```

   Note this won't work on a node that isn't a swarm manager, since secrets get registered in swarm's state database.

2. Alternatively, secret values can be read from a file. In the current directory create a file called `password.txt` and add the value `my-pass` to it. Create a secret with this value:

   ```
   [centos@node-0 ~]$ docker secret create password ./password.txt
   ```


25.3 Managing Secrets

The Docker CLI provides API objects for managing secrets similar to all other Docker assets:

1. List your current secrets:

```
[centos@node-0 ~]$ docker secret ls
```

2. Print secret metadata:

```
[centos@node-0 ~]$ docker secret inspect <secret name>
```

3. Delete a secret:

```
[centos@node-0 ~]$ docker secret rm my-secret
```

25.4 Using Secrets

Secrets are assigned to Swarm services upon creation of the service, and provisioned to containers for that service as they spin up.

1. Create a service authorized to use the `password` secret:

```
[centos@node-0 ~]$ docker service create \
    --name demo \
    --secret password \
    alpine:latest ping 8.8.8.8
```

2. Use `docker service ps demo` to determine what node your service container is running on; ssh into that node, and connect to the container (remember to use `docker container ls` to find the container ID):

```
[centos@node-x ~]$ docker container exec -it <container ID> sh
```

3. Inspect the secrets in this container where they are mounted by default, at `/run/secrets`:

```
/ # cd /run/secrets
/ # ls
/ # cat password
/ # exit
```

   This is the *only* place secret values sit unencrypted in memory.

25.5 Preparing an image for use of secrets

Containers need to consume secrets from their mountpoint, either `/run/secrets` by default, or a custom mount point if defined. In many cases, existing application logic expects secret values to appear behind environment variables; in the following, we set up such a situation as an example.

1. Create a new directory `image-secrets` and navigate to this folder. In this folder create a file named `app.py` and add the following content; this is a Python script that consumes a password from a file with a path specified by the environment variable `PASSWORD_FILE`:

```
import os
print '***** Docker Secrets ******'
print 'USERNAME: {0}'.format(os.environ['USERNAME'])

fname = os.environ['PASSWORD_FILE']
f = open(fname)
try:
    content = f.readlines()
finally:
    f.close()
```



```
print 'PASSWORD_FILE: {0}'.format(fname)
print 'PASSWORD: {0}'.format(content[0])
```

2. Create a file called `Dockerfile` with the following content:

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY . /app
CMD python ./app.py && sleep 1000
```

3. Build the image and push it to a registry so it's available to all nodes in your swarm:

```
[centos@node-0 image-secrets]$ docker image build -t <Docker ID>/secrets-demo:1.0 .
[centos@node-0 image-secrets]$ docker image push <Docker ID>/secrets-demo:1.0
```

4. Create and run a service using this image, and use the `-e` flag to create environment variables that point to your secrets:

```
[centos@node-0 image-secrets]$ docker service create \
    --name secrets-demo \
    --replicas=1 \
    --secret source=password,target=/custom/path/password,mode=0400 \
    -e USERNAME="jdoe" \
    -e PASSWORD_FILE="/custom/path/password" \
    <Docker ID>/secrets-demo:1.0
```

   The `--secret` flag parses a few tokens:

   - `source` indicates the name of the secret (defined when you did `docker secret create...`)
   - `target` indicates the path that the secret file will be mounted at; in this case the secret will be available in a file `/custom/path/password`.
   - `mode` indicates the read/write/execute status for the secret file, with the same octal encoding as `chmod` (so 400 would be read-only for the user who owns the root process in the container, and no access for anyone else).

5. Figure out which node your container is running on, head over there, connect to the container, and run `python app.py`; the `-e` flag in `service create` has set environment variables to point at your secrets, allowing your app to find them where it expects.

25.6 Kubernetes Secrets

Secrets in Kubernetes are manipulated very similarly to Swarm; one interesting difference is the ability to package multiple values or files into the same secret. Below we reproduce the final example from the Swarm section above, but we'll pass in both the username and password in separate files contained in a single Kubernetes secret, rather than passing the username in directly as an environment variable.

1. On your Kubernetes master you set up in the previous exercise, place a username and password in files `username` and `password`:

```
[centos@node-0 ~]$ echo "jdoe" > username
[centos@node-0 ~]$ echo "my-pass" > password
```

2. Create a secret that captures both these files:

```
[centos@node-0 ~]$ kubectl create secret generic user-pass \
    --from-file=./username --from-file=./password
```

   The `generic` keyword here indicates we're going to create the secret from a local file; `user-pass` will be the name of the secret we can refer to later; and the `--from-file` keys list the files we want to include in this secret.

3. Create a pod definition in a file `secretpod.yaml` that uses this secret to map the username file contents directly onto an environment variable, and mount the password file in the container with a second environment variable pointing at its path:

```
apiVersion: v1
kind: Pod
metadata:
  name: secretpod
spec:
  containers:
  - name: democon
    image: <Docker ID>/secrets-demo:1.0
    env:
      - name: USERNAME
        valueFrom:
          secretKeyRef:
            name: user-pass
            key: username
      - name: PASSWORD_FILE
        value: "/custom/path/pass"
    volumeMounts:
    - name: passvol
      mountPath: "/custom/path"
      readOnly: true
  volumes:
  - name: passvol
    secret:
      secretName: user-pass
      items:
      - key: password
        path: pass
  restartPolicy: Never
```

4. Spin the pod up, connect a bash shell to it, check that the environment variables are populated as you'd expect and that the python script works correctly:

```
[centos@node-0 ~]$ kubectl create -f secretpod.yaml
[centos@node-0 ~]$ kubectl exec -it secretpod bash
root@secretpod:/app# echo $USERNAME
root@secretpod:/app# echo $PASSWORD_FILE
root@secretpod:/app# python app.py
```

5. Look in `/custom/path` inside your running container, and notice that only `password` is present, mounted as `pass`. `username` wasn't mentioned in the `items` list for for the `user-pass` secret, and so wasn't mounted in the corresponding volume. In this way you can pick and choose which files from a single secret are mounted into a running container.

6. Compare the config in `secretpod.yaml` to the last `docker service create...` command in the last section. Identify the corresponding pieces of syntax between the Swarm and Kubernetes declarations. Where are environment variables declared? How are secrets associated with the service or pod? How do you point an environment variable at a secret mounted in a file?

25.7 Conclusion

In this lab we have learned how to create, inspect and use secrets in both Swarm and Kubernetes. As is often the case, Swarm's syntax is a bit shorter and simpler, but Kubernetes offers more flexibility and expressiveness, allowing the user to package multiple tokens in a single secret object, and selectively mount them in only the containers that need them.

26 Containerizing an Application

In this exercise, you'll be provided with the application logic of a simple three tier application; your job will be to write Dockerfiles to containerize each tier, and write a Docker Compose file to orchestrate the deployment of that app. This application serves a website that presents cat gifs pulled from a database. The tiers are as follows:

- **Database**: Postgres 9.6
- **API**: Java SpringBoot built via Maven
- **Frontend**: NodeJS + Express

Basic success means writing the Dockerfiles and docker-compose file needed to deploy this application to your orchestrator of choice; to go beyond this, think about minimizing image size, maximizing image performance, and making good choices regarding configuration management.

Start by cloning the source code for this app:

```
[centos@node-0 ~]$ git clone -b ee2.1 \
    https://github.com/docker-training/fundamentals-final.git
```

26.1 Containerizing the Database

1. Navigate to `fundamentals-final/database` to find the config for your database tier.
2. Begin writing a Dockerfile for your postgres database image by choosing an appropriate base image.
3. Your developers have provided you with postgres configuration files `pg_hba.conf` and `postgresql.conf`. Both of these need to be present in `/usr/share/postgresql/9.6/`.
4. You have also been provided with `init-db.sql`. This SQL script populates your database with the required cat gifs. Read the docs for Postgres and / or your base image to determine how to run this script.
5. Postgres expects the environment variables `POSTGRES_USER` and `POSTGRES_DB` to be set at runtime. Make sure these are set to `gordonuser` and `ddev`, respectively, when your database container is running.
6. Once you've built your image, try running it, connecting to it, and querying the postgres database to make sure it's up and running as you'd expect:

   ```
   [centos@node-0 ~]$ docker container run --name database -d mydb:latest
   [centos@node-0 ~]$ docker container exec -it database bash
   [root@641172f742a5 /]# psql ddev gordonuser -c 'select * from images;'
   ```

   If everything is working correctly, you should see a table with URLs to cat gifs returned by the query. Exit and delete this container once you're satisfied that it is working correctly.

26.2 Containerizing the API

1. Navigate to `fundamentals-final/api` to find the source and config for your api tier.
2. We intend to build this SpringBoot API with Maven. Begin writing a Dockerfile for your API by choosing an appropriate base image for your **build** environment.
3. Your developers gave you the following pieces of information:
   - Everything Maven needs to build our API is in `fundamentals-final/api`.
   - The Maven commands to build your API are:

   ```
   $ mvn -B -f pom.xml -s /usr/share/maven/ref/settings-docker.xml dependency:resolve
   $ mvn -B -s /usr/share/maven/ref/settings-docker.xml package -DskipTests
   ```

   - This will produce a jar file `target/ddev-0.0.1-SNAPSHOT.jar` at the path where you ran Maven.
   - In order to successfully access Postgres, the **execution** environment for your API should be based on Java 8 in an alpine environemnt, and have the user `gordon`, as per:


```
$ adduser -Dh /home/gordon gordon
```

- The correct command to launch your API after it's built is:

```
$ java -jar <path to jar file>/ddev-0.0.1-SNAPSHOT.jar \
    --spring.profiles.active=postgres
```

Use this information to finish writing your API Dockerfile. Mind your image size, and think about what components need to be present in production.

4. Once you've built your API image, set up a simple integration test between your database and api by creating a container for each, attached to a network:

```
[centos@node-0 ~]$ docker network create demo_net
[centos@node-0 ~]$ docker container run \
    -d --network demo_net --name database mydb:latest
[centos@node-0 ~]$ docker container run \
    -d --network demo_net -p 8080:8080 --name api myapi:latest
```

5. Curl an API endpoint:

```
[centos@node-0 ~]$ curl localhost:8080/api/pet
```

If everything is working correctly, you should see a JSON response containing one of the cat gif URLs from the database. Leave this integration environment running for now.

26.3 Containerizing the Frontend

1. Navigate to `fundamentals-final/ui` to find the source and config for your web frontend.
2. You know the following about setting up this frontend:
   - It's a node application.
   - The filesystem structure under `fundamentals-final/ui` is exactly as it should be in the frontend's running environment.
   - Install proceeds by running `npm install` in the same directory as `package.json`.
   - The frontend is started by running `node src/server.js`.

   Write a Dockerfile that makes an appropriate environment, installs the frontend and starts it on container launch.
3. Once you've built your ui image, start a container based on it, and attach it to your integration environment from the last step. Check to see if you can hit your website in your browser at `IP:port/pet`; if so, you have successfully containerized all three tiers of your application.

26.4 Orchestrating the Application

Once all three elements of the application are containerized, it's time to assemble them into a functioning application by writing a Docker compose file. The environmental requirements for each service are as follows:

- **Database**:
- Named `database`.
- Make sure the environment variables `POSTGRES_USER` and `POSTGRES_DB` are set in the compose file, if they weren't set in the database's Dockerfile (when would you want to set them in one place versus the other?).
- The database will need to communicate with the API.
- **API**:
- Named `api`.
- The API needs to communicate with both the database and the web frontend.
- **Frontend**:
- Named `ui`.

- Serves the web frontend on container port 3000.
- Needs to be able to communicate with the API.

Write a `docker-compose.yml` to capture this configuration, and use it to stand up your app with Docker Compose, Swarm, or Kubernetes. Make sure the website is reachable from the browser.

26.5 Conclusion

In this exercise, you containerized and orchestrated a simple three tier application by writing a Dockerfile for each service, and a Docker Compose file for the full application. In practice, developers should be including their Dockerfiles with their source code, and senior developers and / or application architects should be providing Docker Compose files for the full application, possibly in conjunction with the operations team for environment-specific config.

Compare your Dockerfiles and Docker Compose file with other people in the class; how do your solutions differ? What are the possible advantages of each approach?

27 Cleaning up Docker Resources

By the end of this exercise, you should be able to:

- Assess how much disk space docker objects are consuming
- Use `docker prune` commands to clear out unneeded docker objects
- Apply label based filters to `prune` commands to control what gets deleted in a cleanup operation

1. Find out how much memory Docker is using by executing:

   ```
   [centos@node-3 ~]$ docker system df
   ```

   The output will show us how much space images, containers and local volumes are occupying and how much of this space can be reclaimed.

2. Reclaim all reclaimable space by using the following command:

   ```
   [centos@node-3 ~]$ docker system prune
   ```

   Answer with `y` when asked if we really want to remove all unused networks, containers, images and volumes.

3. Create a couple of containers with labels (these will exit immediately; why?):

   ```
   [centos@node-3 ~]$ docker container run --label apple --name fuji -d alpine
   [centos@node-3 ~]$ docker container run --label orange --name clementine -d alpine
   ```

4. Delete only those stopped containers bearing the `apple` label:

   ```
   [centos@node-3 ~]$ docker container ls -a
   [centos@node-3 ~]$ docker container prune --filter 'label=apple'
   [centos@node-3 ~]$ docker container ls -a
   ```

   Only the container named `clementine` should remain after the targeted prune.

5. Finally, prune containers launched before a given timestamp using the `until` filter; start by getting the current RFC 3339 time (https://tools.ietf.org/html/rfc3339 - note Docker *requires* the otherwise optional `T` separating date and time), then creating a new container:

   ```
   [centos@node-3 ~]$ TIMESTAMP=$(date --rfc-3339=seconds | sed 's/ /T/')
   [centos@node-3 ~]$ docker container run --label tomato --name beefsteak -d alpine
   ```

   And use the timestamp returned in a prune:

   ```
   [centos@node-3 ~]$ docker container prune -f --filter "until=$TIMESTAMP"
   [centos@node-3 ~]$ docker container ls -a
   ```


Note the `-f` flag, to suppress the confirmation step. `label` and `until` filters for pruning are also available for networks and images, while data volumes can only be selectively pruned by `label`; finally, images can also be pruned by the boolean `dangling` key, indicating if the image is untagged.

27.1 Conclusion

In this exercise, we saw some very basic `docker prune` usage - most of the top-level docker objects have a `prune` command (`docker container prune`, `docker volume prune` etc). Most docker objects leave something on disk even after being shut down; consider using these cleanup commands as part of your cluster maintenance and garbage collection plan, to avoid accidentally running out of disk on your Docker hosts.

28 Inspection Commands

By the end of this exercise, you should be able to:

- Gather system level info from the docker engine
- Consume and format the docker engine's event stream for monitoring purposes

28.1 Inspecting System Information

1. We can find the `info` command under `system`. Execute:

   ```
   [centos@node-3 ~]$ docker system info
   ```

   This provides some high-level information about the docker deployment on the current node, and the node itself. From this output, identify:

   - how many images are cached on your machine?
   - how many containers are running or stopped?
   - what version of containerd are you running?
   - whether Docker is running in swarm mode?

28.2 Monitoring System Events

1. There is another powerful system command that allows us to monitor what's happening on the Docker host. Execute the following command:

   ```
   [centos@node-3 ~]$ docker system events
   ```

   Please note that it looks like the system is hanging, but that is not the case. The system is just waiting for some events to happen.

2. Open a second connection to `node-3` and execute the following command:

   ```
   [centos@node-3 ~]$ docker container run --rm alpine echo 'Hello World!'
   ```

   and observe the generated output in the first terminal. It should look similar to this:

   ```
   2017-01-25T16:57:48.553596179-06:00 container create 30eb63 ...
   2017-01-25T16:57:48.556718161-06:00 container attach 30eb63 ...
   2017-01-25T16:57:48.698190608-06:00 network connect de1b2b ...
   2017-01-25T16:57:49.062631155-06:00 container start 30eb63 ...
   2017-01-25T16:57:49.065552570-06:00 container resize 30eb63 ...
   2017-01-25T16:57:49.164526268-06:00 container die 30eb63 ...
   2017-01-25T16:57:49.613422740-06:00 network disconnect de1b2b ...
   2017-01-25T16:57:49.815845051-06:00 container destroy 30eb63 ...
   ```

   Granular information about every action taken by the Docker engine is presented in the events stream.

3. If you don't like the format of the output then we can use the `--format` parameter to define our own format in the form of a Go template. Stop the events watch on your first terminal with `CTRL+C`, and try this:

```
[centos@node-3 ~]$ docker system events --format '--> {{.Type}}-{{.Action}}'
```

now the output looks a little bit less cluttered when we run our alpine container on the second terminal as above.

4. Finally we can find out what the event structure looks like by outputting the events in `json` format (once again after killing the events watcher on the first terminal and restarting it with):

```
[centos@node-3 ~]$ docker system events --format '{{json .}}' | jq
```

which should give us for the first event in the series after re-running our alpine container on the other connection to `node-3` something like this (note, the output has been prettyfied for readability):


```
{
   "status":"create",
   "id":"95ddb6ed4c87d67fa98c3e63397e573a23786046e00c2c68a5bcb9df4c17635c",
   "from":"alpine",
   "Type":"container",
   "Action":"create",
   "Actor":{
      "ID":"95ddb6ed4c87d67fa98c3e63397e573a23786046e00c2c68a5bcb9df4c17635c",
      "Attributes":{
         "image":"alpine",
         "name":"sleepy_roentgen"
      }
   },
   "time":1485385702,
   "timeNano":1485385702748011034
}
```


28.3 Conclusion

In this exercise we have learned how to inspect system wide properties of our Docker host by using the `docker system info` command; this is one of the first places to look for general config information to include in a bug report. We also saw a simple example of `docker system events`; the events stream is one of the primary sources of information that should be logged and monitored when running Docker in production. Many commercial as well as open source products (such as Elastic Stack) exist to facilitate aggregating and mining these streams at scale.

29 Plugins

By the end of this exercise, you should be able to:

- Install, configure, and delete any Docker plugin
- Use the `vieux/sshfs` plugin to create ssh-mountable volumes that can be mounted into any container in your cluster

29.1 Installing a Plugin

1. Plugins can be hosted on Docker Hub or any other (private) repository. Let's start with Docker Hub. Browse to https://hub.docker.com and enter `vieux/sshfs` in the search box. The result should show you the plugin that we are going to work with.
2. Install the plugin into our Docker Engine:


```
[centos@node-0 ~]$ docker plugin install vieux/sshfs
```

The system should ask us for permission to use privileges. In the case of the `sshfs` plugin there are 4 privileges. Answer with `y`.

3. Once we have successfully installed some plugins we can use the `ls` command to see the status of each of the installed plugins. Execute:

```
[centos@node-0 ~]$ docker plugin ls
```

29.2 Enabling and Disabling a Plugin

1. Once a plugin is installed it is `enabled` by default. We can disable it using this command:

```
[centos@node-0 ~]$ docker plugin disable vieux/sshfs
```

only when a plugin is disabled can certain operations on it be executed.

2. The plugin can be (re-) enabled by using this command:

```
[centos@node-0 ~]$ docker plugin enable vieux/sshfs
```

Play with the above commands and notice how the status of the plugin changes when displaying it with `docker plugin ls`.

29.3 Inspecting a Plugin

1. We can also use the `inspect` command to further inspect all the attributes of a given plugin. Execute the following command:

```
[centos@node-0 ~]$ docker plugin inspect vieux/sshfs
```

and examine the output. Specifically note that there are two sections in the metadata called `Env`, one is under `Config` and the other under `Settings`. This is where the list of environment variables are listed that the author of the plugin has defined. In this specific situation we can see that there is a single variable called `DEBUG` defined. Its initial value is `0`.

2. We can use the `set` command to change values of the environment variables. Execute:

```
[centos@node-0 ~]$ docker plugin set vieux/sshfs DEBUG=1

Error response from daemon: cannot set on an active plugin, disable plugin before setting
```

This is one of those times we have to disable the plugin first; do so, then try the `set` command again:

```
[centos@node-0 ~]$ docker plugin disable vieux/sshfs
[centos@node-0 ~]$ docker plugin set vieux/sshfs DEBUG=1
[centos@node-0 ~]$ docker plugin enable vieux/sshfs
```

and then inspect again the metadata of the plugin. Notice how the value of `DEBUG` has been adjusted. Only the one under the `Settings` node changed but the one under the `Config` node still shows the original (default) value.

29.4 Using the Plugin

1. Make a directory on `node-1` that we will mount as a volume across our cluster:

```
[centos@node-1 ~]$ mkdir ~/demo
```

2. Back on `node-0`, use the plugin to create a volume that can be mounted via ssh:


```
[centos@node-0 ~]$ docker volume create -d vieux/sshfs \
    -o sshcmd=centos@<node-1 public IP>:/home/centos/demo \
    -o password=orca \
    sshvolume
```

3. Mount that volume in a new container as per usual:

```
[centos@node-0 ~]$ docker container run --rm -it -v sshvolume:/data alpine sh
```

4. Inside the container navigate to the `/data` folder and create a new file:

```
/ # cd /data
/ # echo 'Hello from client!' > demo.txt
/ # ls -al
```

5. Head over to `node-1`, and confirm that `demo.txt` got written there.

29.5 Removing a Plugin

1. If we don't want or need this plugin anymore we can remove it using the command:

```
[centos@node-0 ~]$ docker volume rm sshvolume
[centos@node-0 ~]$ docker plugin disable vieux/sshfs
[centos@node-0 ~]$ docker plugin rm vieux/sshfs
```

   Note how we first have to disable the plugin before we can remove it.

29.6 Conclusion

Docker follows a 'batteries included but swappable' mindset in its product design: everything you need to get started is included, but heavy customization is supported and encouraged. Docker plugins are one aspect of that flexibility, allowing users to define their own volume and networking behavior.

Instructor Demos

1 Instructor Demo: Process Isolation

In this demo, we'll illustrate:

- What containerized process IDs look like inside versus outside of a kernel namespace
- How to impose control group limitations on CPU and memory consumption of a containerized process.

1.1 Exploring the PID Kernel Namespace

1. Start a simple container we can explore:

```
[centos@node-0 ~]$ docker container run -d --name pinger centos:7 ping 8.8.8.8
```

2. Use `docker container exec` to launch a child process inside the container's namespaces:

```
[centos@node-0 ~]$ docker container exec pinger ps -aux

USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.1  0.0  24860  1884 ?        Ss   02:20   0:00 ping 8.8.8.8
root          5  0.0  0.0  51720  3504 ?        Rs   02:20   0:00 ps -aux
```


3. Run the same ps directly on the host, and search for your ping process:

```
[centos@node-0 ~]$ ps -aux | grep ping

USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     11622  0.0  0.0  24860  1884 ?        Ss   02:20   0:00 ping 8.8.8.8
centos   11839  0.0  0.0 112656  2132 pts/0    S+   02:23   0:00 grep --color=auto ping
```

The ping process appears as PID 1 inside the container, but as some higher PID (11622 in this example) from outside the container.

4. List your containers to show this ping container is still running:

```
[centos@node-0 ~]$ docker container ls

CONTAINER ID  IMAGE     COMMAND         ...  STATUS        ...  NAMES
bb3a3b1cbb78  centos:7  "ping 8.8.8.8"  ...  Up 6 minutes       pinger
```

Kill the ping process by host PID, and show the container has stopped:

```
[centos@node-0 ~]$ sudo kill -9 [host PID of ping]
[centos@node-0 ~]$ docker container ls

CONTAINER ID  IMAGE     COMMAND         ...  STATUS        ...  NAMES
```

Killing the ping process on the host also kills the container - all a running container is is its PID 1 process, and the kernel tooling that isolates it from the host. Note using `kill -9` is just for demonstration purposes here; never stop containers this way.

1.2 Imposing Resource Limitations With Cgroups

1. Start a container that consumes two full CPUs:

```
[centos@node-0 ~]$ docker container run -d training/stress:2.1 --vm 2
```

Here the `--vm` flag starts 2 dummy processes that allocate and free memory as fast as they can, each consuming as many CPU cycles as possible.

2. Check the CPU consumption of processes in the container:

```
[centos@node-0 ~]$ docker container top <container ID>

UID     PID     PPID    C   ...   CMD
root    5806    5789    0   ...   /usr/bin/stress --verbose --vm 2
root    5828    5806    99  ...   /usr/bin/stress --verbose --vm 2
root    5829    5806    99  ...   /usr/bin/stress --verbose --vm 2
```

That `C` column represents CPU consumption, in percent; this container is hogging two full CPUs! See the same thing by running `ps -aux` both inside and outside this container, like we did above; the same process and its CPU utilization is visible inside and outside the container:

```
[centos@node-0 ~]$ docker container exec <container ID> ps -aux

USER       PID %CPU %MEM    ...   COMMAND
root         1  0.0  0.0    ...   /usr/bin/stress --verbose --vm 2
root         5 98.9  6.4    ...   /usr/bin/stress --verbose --vm 2
root         6 99.0  0.4    ...   /usr/bin/stress --verbose --vm 2
root         7  2.0  0.0    ...   ps -aux
```

And on the host directly, via the PIDs we found from `docker container top` above:


```
[centos@node-0 ~]$ ps -aux | grep <PID>

USER        PID %CPU %MEM    ...   COMMAND
root       5828 99.3  4.9    ...   /usr/bin/stress --verbose --vm 2
centos     6327  0.0  0.0    ...   grep --color=auto 5828
```

3. Kill off this container:

```
[centos@node-0 ~]$ docker container rm -f <container ID>
```

This is the right way to kill and remove a running container (not `kill -9`).

4. Run the same container again, but this time with a cgroup limitation on its CPU consumption:

```
[centos@node-0 ~]$ docker container run -d --cpus="1" training/stress:2.1 --vm 2
```

Do `docker container top` and `ps -aux` again, just like above; you'll see the processes taking up half a CPU each, for a total of 1 CPU consumed. The `--cpus="1"` flag has imposed a control group limitation on the processes in this container, constraining them to consume a total of no more than one CPU.

5. Find the host PID of a process running in this container using `docker container top` again, and then see what cgroups that process lives in on the host:

```
[centos@node-0 ~]$ cat /proc/<host PID of containerized process>/cgroup

12:memory:/docker/31d03...
11:freezer:/docker/31d03...
10:hugetlb:/docker/31d03...
9:perf_event:/docker/31d03...
8:net_cls,net_prio:/docker/31d03...
7:cpuset:/docker/31d03...
6:pids:/docker/31d03...
5:blkio:/docker/31d03...
4:rdma:/
3:devices:/docker/31d03...
2:cpu,cpuacct:/docker/31d03...
1:name=systemd:/docker/31d03...
```

6. Get a summary of resources consumed by processes in a control group via `systemd-cgtop`:

```
[centos@node-0 ~]$ systemd-cgtop

Path                Tasks   %CPU     Memory  Input/s    Output/s

/                   68      112.3    1.0G    -          -
/docker             -       99.3     301.0M  -          -
/docker/31d03...    3       99.3     300.9M  -          -
...
```

Here again we can see that the processes living in the container's control group (`/docker/31d03...`) are constrained to take up only about 1 CPU.

7. Remove this container, spin up a new one that creates a lot of memory pressure, and check its resource consumption with `docker stats`:

```
[centos@node-0 ~]$ docker container rm -f <container ID>
[centos@node-0 ~]$ docker container run -d training/stress:2.1 --vm 2 --vm-bytes 1024M
[centos@node-0 ~]$ docker stats

CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %    ...
b29a6d877343        198.94%             937.2MiB / 3.854GiB   23.75%   ...
```


8. Kill this container off, start it again with a memory constraint, and list your containers:

```
[centos@node-0 ~]$ docker container rm -f <container ID>
[centos@node-0 ~]$ docker container run \
    -d -m 256M training/stress:2.1 --vm 2 --vm-bytes 1024M
[centos@node-0 ~]$ docker container ls -a

CONTAINER ID        IMAGE                 ...  STATUS
296c8f76af5c        training/stress:2.1   ...  Exited (1) 26 seconds ago
```

   It exited immediately this time.

9. Inspect the metadata for this container, and look for the `OOMKilled` key:

```
[centos@node-0 ~]$ docker container inspect <container ID> | grep 'OOMKilled'

        "OOMKilled": true,
```

   When the containerized process tried to exceed its memory limitation, it gets killed with an Out Of Memory exception.

1.3 Conclusion

In this demo, we explored some of the most important technologies that make containerization possible: kernel namespaces and control groups. The core message here is that containerized processes are just processes running on their host, isolated and constrained by these technologies. All the tools and management strategies you would use for conventional processes apply just as well for containerized processes.

2 Instructor Demo: Creating Images

In this demo, we'll illustrate:

- How to read each step of the image build output
- How intermediate image layers behave in the cache and as independent images
- What the meanings of 'dangling' and `<missing>` image layers are

2.1 Understanding Image Build Output

1. Make a folder `demo` for our image demo:

```
[centos@node-0 ~]$ mkdir demo ; cd demo
```

   And create a Dockerfile therein with the following content:

```
FROM centos:7
RUN yum update -y
RUN yum install -y which
RUN yum install -y wget
RUN yum install -y vim
```

2. Build your image from your Dockerfile, just like we did in the last exercise:

```
[centos@node-0 demo]$ docker image build -t demo .
```

3. Examine the output from the build process. The very first line looks like:

```
Sending build context to Docker daemon  2.048kB
```


Here the Docker daemon is archiving everything at the path specified in the `docker image build` command (`.` or the current directory in this example). This is why we made a fresh directory `demo` to build in, so that nothing extra is included in this process.

4. The next lines look like:

```
Step 1/5 : FROM centos:7
 ---> 49f7960eb7e4
```

Do an `image ls`:

```
[centos@node-0 demo]$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
demo                latest              59e595750dd5        10 seconds ago      645MB
centos              7                   49f7960eb7e4        2 months ago        200MB
```

Notice the Image ID for `centos:7` matches that second line in the build output. The build starts from the base image defined in the `FROM` command.

5. The next few lines look like:

```
Step 2/5 : RUN yum update -y
 ---> Running in 8734b14cf011
Loaded plugins: fastestmirror, ovl
...
```

This is the output of the `RUN` command, `yum update -y`. The line `Running in 8734b14cf011` specifies a container that this command is running in, which is spun up based on all previous image layers (just the `centos:7` base at the moment). Scroll down a bit and you should see something like:

```
 ---> 433e56d735f6
Removing intermediate container 8734b14cf011
```

At the end of this first `RUN` command, the temporary container `8734b14cf011` is saved as an image layer `433e56d735f6`, and the container is removed. This is the exact same process as when you used `docker container commit` to save a container as a new image layer, but now running automatically as part of a Dockerfile build.

6. Look at the history of your image:

```
[centos@node-0 demo]$ docker image history demo

IMAGE               CREATED             CREATED BY                                      SIZE
59e595750dd5        2 minutes ago       /bin/sh -c yum install -y vim                   142MB
bba17f8df167        2 minutes ago       /bin/sh -c yum install -y wget                  87MB
b9f2efa616de        2 minutes ago       /bin/sh -c yum install -y which                 86.6MB
433e56d735f6        2 minutes ago       /bin/sh -c yum update -y                        129MB
49f7960eb7e4        2 months ago        /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>           2 months ago        /bin/sh -c #(nop)  LABEL org.label-schema....   0B
<missing>           2 months ago        /bin/sh -c #(nop) ADD file:8f4b3be0c1427b1...   200MB
```

As you can see, the different layers of `demo` correspond to a separate line in the Dockerfile and the layers have their own ID. You can see the image layer `433e56d735f6` committed in the second build step in the list of layers for this image.

7. Look through your build output for where steps 3/5 (installing `which`), 4/5 (installing `wget`), and 5/5 (installing `vim`) occur - the same behavior of starting a temporary container based on the previous image layers, running the `RUN` command, saving the container as a new image layer visible in your `docker iamge history` output, and deleting the temporary container is visible.

8. Every layer can be used as you would use any image, which means we can inspect a single layer. Let's inspect the `wget` layer, which in my case is `bba17f8df167` (yours will be different, look at your `docker image history` output):


```
[centos@node-0 demo]$ docker image inspect bba17f8df167
```

9. Let's look for the command associated with this image layer by using `--format`:

```
[centos@node-0 demo]$ docker image inspect \
    --format='{{.ContainerConfig.Cmd}}' bba17f8df167

[/bin/sh -c yum install -y wget]
```

10. We can even start containers based on intermediate image layers; start an interactive container based on the `wget` layer, and look for whether `wget` and `vim` are installed:

```
[centos@node-0 demo]$ docker container run -it bba17f8df167 bash
[root@a766a3d616b7 /]# which wget
/usr/bin/wget

[root@a766a3d616b7 /]# which vim
/usr/bin/which: no vim in (/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin)
```

`wget` is installed in this layer, but since `vim` didn't arrive until the next layer, it's not available here.

2.2 Managing Image Layers

1. Change the last line in the Dockerfile from the last section to install `nano` instead of `vim`:

```
FROM centos:7
RUN yum update -y
RUN yum install -y which
RUN yum install -y wget
RUN yum install -y nano
```

2. Rebuild your image, and list your images again:

```
[centos@node-0 demo]$ docker image build -t demo .
[centos@node-0 demo]$ docker image ls

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
demo                latest              5a6aedc1feab        8 seconds ago       590MB
<none>              <none>              59e595750dd5        23 minutes ago      645MB
centos              7                   49f7960eb7e4        2 months ago        200MB
```

What is that image named `<none>`? Notice the image ID is the same as the old image ID for `demo:latest` (see your history output above). The name and tag of an image is just a pointer to the stack of layers that make it up; reuse a name and tag, and you are effectively moving that pointer to a new stack of layers, leaving the old one (the one containing the `vim` install in this case) as an untagged or 'dangling' image.

3. Rewrite your Dockerfile one more time, to combine some of those install steps:

```
FROM centos:7
RUN yum update -y
RUN yum install -y which wget nano
```

Rebuild using a `new` tag this time, and list your images one more time:

```
[centos@node-0 demo]$ docker image build -t demo:new .
...
[centos@node-0 demo]$ docker image ls

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
demo                new                 568b29a0dce9        20 seconds ago      416MB
```



```
demo                latest              5a6aedc1feab        5 minutes ago       590MB
<none>              <none>              59e595750dd5        28 minutes ago      645MB
centos              7                   49f7960eb7e4        2 months ago        200MB
```

Image `demo:new` is much smaller in size than `demo:latest`, even though it contains the exact same software - why?

2.3 Conclusion

In this demo, we explored the layered structure of images; each layer is built as a distinct image and can be treated as such, on the host where it was built. This information is preserved on the build host for use in the build cache; build another image based on the same lower layers, and they will be reused to speed up the build process. Notice that the same is not true of downloaded images like `centos:7`; intermediate image caches are not downloaded, but rather only the final complete image.

3 Instructor Demo: Basic Volume Usage

In this demo, we'll illustrate:

- Creating, updating, destroying, and mounting docker named volumes
- How volumes interact with a container's layered filesystem
- Usecases for mounting host directories into a container

3.1 Using Named Volumes

1. Create a volume, and inspect its metadata:

```
[centos@node-0 ~]$ docker volume create demovol
[centos@node-0 ~]$ docker volume inspect demovol

[
    {
        "CreatedAt": "2018-11-03T19:07:56Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/demovol/_data",
        "Name": "demovol",
        "Options": {},
        "Scope": "local"
    }
]
```

   We can see that by default, named volumes are created under `/var/lib/docker/volumes/<name>/_data`.

2. Run a container that mounts this volume, and list the filesystem therein:

```
[centos@node-0 ~]$ docker container run -it -v demovol:/demo centos:7 bash
[root@f4aca1b60965 /]# ls
anaconda-post.log  bin  demo  dev  etc  home ...
```

   The `demo` directory is created as the mountpoint for our volume, as specified in the flag `-v demovol:/demo`. This should also appear in your container filesystem's list of mountpoints:

```
[root@f4aca1b60965 /]# cat /proc/self/mountinfo | grep demo

1199 1180 202:1 /var/lib/docker/volumes/demovol/_data /demo rw,relatime - xfs /dev/xvda1 ...
```


3. Put a file in this volume:

```
[root@f4aca1b60965 /]# echo 'dummy file' > /demo/mydata.dat
```

4. Exit the container, and list the contents of your volume on the host:

```
[centos@node-0 ~]$ sudo ls /var/lib/docker/volumes/demovol/_data
```

You'll see your `mydata.dat` file present at this point in the host's filesystem. Delete the container:

```
[centos@node-0 ~]$ docker container rm -f <container ID>
```

The volume and its contents will still be present on the host.

5. Start a new container mounting the same volume, attach a bash shell to it, and show that the old data is present in your new container:

```
[centos@node-0 ~]$ docker container run -d -v demovol:/demo centos:7 ping 8.8.8.8
[centos@node-0 ~]$ docker container exec -it <container ID> bash
[root@11117d3de672 /]# cat /demo/mydata.dat
```

6. Exit this container, and inspect its mount metadata:

```
[centos@node-0 ~]$ docker container inspect <container ID>

    "Mounts": [
        {
            "Type": "volume",
            "Name": "demovol",
            "Source": "/var/lib/docker/volumes/demovol/_data",
            "Destination": "/demo",
            "Driver": "local",
            "Mode": "z",
            "RW": true,
            "Propagation": ""
        }
    ],
```

Here too we can see the volumes and host mountpoints for everything mounted into this container.

7. Build a new image out of this container using `docker container commit`, and start a new container based on that image:

```
[centos@node-0 ~]$ docker container commit <container ID> demo:snapshot
[centos@node-0 ~]$ docker container run -it demo:snapshot bash
[root@ad62f304ba18 /]# cat /demo/mydata.dat
cat: /demo/mydata.dat: No such file or directory
```

The information mounted into the original container is not part of the container's layered filesystem, and therefore is not captured in the image creation process; volume mounts and the layered filesystem are completely separate.

8. Clean up by removing that volume:

```
[centos@node-0 ~]$ docker volume rm demovol
```

You will get an error saying the volume is in use - docker will not delete a volume mounted to any container (even a stopped container) in this way. Remove the offending container first, then remove the volume again.

3.2 Mounting Host Paths

1. Make a directory with some source code in it for your new website:


```
[centos@node-0 ~]$ mkdir /home/centos/myweb
[centos@node-0 ~]$ cd /home/centos/myweb
[centos@node-0 myweb]$ echo "<h1>Hello Wrld</h1>" > index.html
```

2. Start up an nginx container that mounts this as a static website:

```
[centos@node-0 myweb]$ docker container run -d \
    -v /home/centos/myweb:/usr/share/nginx/html \
    -p 8000:80 nginx
```

   Visit your website at the public IP of this node, port 8000.

3. Fix the spelling of 'world' in your HTML, and refresh the webpage; the content served by nginx gets updated without having to restart or replace the nginx container.

3.3 Conclusion

In this demo, we saw two key points about volumes: they exist outside the container's layered filesystem, meaning that not only are they not captured on image creation, they don't participate in the usual copy on write procedure when manipulating files in the writable container layer. Second, we saw that manipulating files on the host that have been mounted into a container immediately propagates those changes to the running container; this is a popular technique for developers who containerize their running environment, and mount in their in-development code so they can edit their code using the tools on their host machine that they are familiar with, and have those changes immediately available inside a running container without having to restart or rebuild anything.

4 Instructor Demo: Single Host Networks

In this demo, we'll illustrate:

- Creating docker bridge networks
- Attaching containers to docker networks
- Inspecting networking metadata from docker networks and containers
- How network interfaces appear in different network namespaces
- What network interfaces are created on the host by docker networking
- What iptables rules are created by docker to isolate docker software-defined networks and forward network traffic to containers

4.1 Following Default Docker Networking

1. On a fresh node you haven't run any containers on yet, list your networks:

```
[centos@node-1 ~]$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
7c4e63830cbf        bridge              bridge              local
c87d2a849036        host                host                local
902af00d5511        none                null                local
```

2. Get some metadata about the `bridge` network, which is the default network containers attach to when doing `docker container run`:

```
[centos@node-1 ~]$ docker network inspect bridge
```

   Notice the `IPAM` section:

```
"IPAM": {
    "Driver": "default",
```



```
    "Options": null,
    "Config": [
        {
            "Subnet": "172.17.0.0/16",
            "Gateway": "172.17.0.1"
        }
    ]
}
```

Docker's IP address management driver assigns a subnet (`172.17.0.0/16` in this case) to each bridge network, and uses the first IP in that range as the network's gateway.

Also note the `containers` key:

```
"Containers": {}
```

So far, no containers have been plugged into this network.

3. Have a look at what network interfaces are present on this host:

```
[centos@node-1 ~]$ ip addr

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP qlen 1000
    link/ether 12:eb:dd:4e:07:ec brd ff:ff:ff:ff:ff:ff
    inet 10.10.17.74/20 brd 10.10.31.255 scope global dynamic eth0
       valid_lft 2444sec preferred_lft 2444sec
    inet6 fe80::10eb:ddff:fe4e:7ec/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:e2:c5:a4:6b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
```

We see the usual `eth0` and loopback interfaces, but also the `docker0` linux bridge, which corresponds to the docker software defined network we were inspecting in the previous step; note it has the same gateway IP as we found when doing `docker network inspect`.

4. Create a docker container without specifying any networking parameters, and do the same `docker network inspect` as above:

```
[centos@node-1 ~]$ docker container run -d centos:7 ping 8.8.8.8
[centos@node-1 ~]$ docker network inspect bridge

...
"Containers": {
    "f4e8f3f1b918900dd8c9b8867aa3c81e95cf34aba7e366379f2a9ade9987a40b": {
        "Name": "zealous_kirch",
        "EndpointID": "f9f246aaff3d2b62556949b54842937871e17dcd40a0986ed8b78008408ccb5f",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    }
}
...
```


The `Containers` key now contains the metadata for the container you just started; it received the next available IP address from the default network's subnet. Also note that the last four digits of the container's MAC address are the same as its IP on this network - this encoding ensures containers get a locally unique MAC address that linux bridges can route traffic to.

5. Look at your network interfaces again:

```
[centos@node-1 ~]$ ip addr

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP qlen 1000
    link/ether 12:eb:dd:4e:07:ec brd ff:ff:ff:ff:ff:ff
    inet 10.10.17.74/20 brd 10.10.31.255 scope global dynamic eth0
       valid_lft 2188sec preferred_lft 2188sec
    inet6 fe80::10eb:ddff:fe4e:7ec/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:e2:c5:a4:6b brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:e2ff:fec5:a46b/64 scope link
       valid_lft forever preferred_lft forever
5: vethfbd45f0@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state U
    link/ether 6e:3c:e4:21:7b:e2 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::6c3c:e4ff:fe21:7be2/64 scope link
       valid_lft forever preferred_lft forever
```

A new interface has appeared: interface number 5 is the veth connection connecting the container's network namespace to the host's network namespace. But, what happened to interface number 4? It's been skipped in the list.

Look closely at interface number 5:

```
5: vethfbd45f0@if4
```

That `@if4` indicates that interface number 5 is connected to interface 4. In fact, these are the two endpoints of the veth connection mentioned above; each end of the connection appears as a distinct interface, and `ip addr` only lists the interfaces in the current network namespace (the host in the above example).

6. Look at the interfaces in your container's network namespace (you'll first need to connect to the container and install `iproute`):

```
[centos@node-1 ~]$ docker container exec -it <container ID> bash
[root@f4e8f3f1b918 /]# yum install -y iproute
...
[root@f4e8f3f1b918 /]# ip addr

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
```


Not only does interface number 4 appear inside the container's network namespace connected to interface 5, but we can see that this veth endpoint inside the container is getting treated as the `eth0` interface inside the container.

4.2 Establishing Custom Docker Networks

1. Create a custom bridge network:

```
[centos@node-1 ~]$ docker network create my_bridge
[centos@node-1 ~]$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
7c4e63830cbf        bridge              bridge              local
c87d2a849036        host                host                local
a04d46bb85b1        my_bridge           bridge              local
902af00d5511        none                null                local
```

`my_bridge` gets created as another linux bridge-based network by default.

2. Run a couple of containers named c2 and c3 attached to this new network:

```
[centos@node-1 ~]$ docker container run \
    --name c2 --network my_bridge -d centos:7 ping 8.8.8.8
[centos@node-1 ~]$ docker container run \
    --name c3 --network my_bridge -d centos:7 ping 8.8.8.8
```

3. Inspect your new bridge:

```
[centos@node-1 ~]$ docker network inspect my_bridge

...
"IPAM": {
    "Driver": "default",
    "Options": {},
    "Config": [
        {
            "Subnet": "172.18.0.0/16",
            "Gateway": "172.18.0.1"
        }
    ]
},
...
"Containers": {
    "084caf415784fb4d58dc6fb4601321114b93dc148793fd66c95fc2c9411b085e": {
        "Name": "c3",
        "EndpointID": "804600568d5c865dc864354ef8dab944131be43f5be1886211e6acd39c3f4801",
        "MacAddress": "02:42:ac:12:00:03",
        "IPv4Address": "172.18.0.3/16",
        "IPv6Address": ""
    },
    "23d2e307325ec022ce6b08406bfb0f7e307fa533a7a4957a6d476c170d8e8658": {
        "Name": "c2",
        "EndpointID": "730ac71839550b960629bf74dda2a9493c8d272ea7db976c3e97f28fedbb5317",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
    }
},
```



```
...
```

The next subnet in sequence (172.18.0.0/16 in my case) has been assigned to my_bridge by the IPAM driver, and containers attached to this network get IPs from this range exactly as they did with the default bridge network.

4. Try to contact container c3 from c2:

```
[centos@node-1 ~]$ docker container exec c2 ping c3
```

It works - containers on the same custom network are able to resolve each other via **DNS lookup of container names**. This means that our application logic (c2 `ping` c3 in this simple case) doesn't have to do any of its own service discovery; all we need to know are container names, and docker does the rest.

5. Start another container on my_bridge, but don't name it:

```
[centos@node-1 ~]$ docker container run --network my_bridge -d centos:7 ping 8.8.8.8
[centos@node-1 ~]$ docker container ls

CONTAINER ID        IMAGE     ... STATUS              PORTS               NAMES
625cb95b922d        centos:7  ... Up 2 seconds                            competent_leavitt
084caf415784        centos:7  ... Up 5 minutes                            c3
23d2e307325e        centos:7  ... Up 5 minutes                            c2
f4e8f3f1b918        centos:7  ... Up 21 minutes                           zealous_kirch
```

As usual, it got a default name generated for it (competent_leavitt in my case). Try resolving this name by DNS as above:

```
[centos@node-1 ~]$ docker container exec c2 ping competent_leavitt

ping: competent_leavitt: Name or service not known
```

DNS resolution fails. Containers must be explicitly named in order to appear in docker's DNS tables.

6. Find the IP of your latest container (competent_leavitt in my case) via `docker container inspect`, and ping it from c2 directly by IP:

```
[centos@node-1 ~]$ docker network inspect my_bridge

...
"625cb95b922d2502fd016c6517c51652e84f902f69632d5d399dc38f3f7b2711": {
    "Name": "competent_leavitt",
    "EndpointID": "2fdb093d97b23da43023b07338a329180995fc0564ed0762147c8796380c51e7",
    "MacAddress": "02:42:ac:12:00:04",
    "IPv4Address": "172.18.0.4/16",
    "IPv6Address": ""
}
...

[centos@node-1 ~]$ docker container exec c2 ping 172.18.0.4

PING 172.18.0.4 (172.18.0.4) 56(84) bytes of data.
64 bytes from 172.18.0.4: icmp_seq=1 ttl=64 time=0.083 ms
64 bytes from 172.18.0.4: icmp_seq=2 ttl=64 time=0.060 ms
```

The ping succeeds. While the default-named container isn't resolvable by DNS, it is still reachable on the my_bridge network.

7. Finally, create container c1 attached to the default network:

```
[centos@node-1 ~]$ docker container run --name c1 -d centos:7 ping 8.8.8.8
```

Attempt to ping it from c2 by name:


```
[centos@node-1 ~]$ docker container exec c2 ping c1

ping: c1: Name or service not known
```

DNS resolution is scoped to user-defined docker networks. Find c1's IP manually as above (mine is at 172.17.0.3), and ping this IP directly from c2:

```
[centos@node-1 ~]$ docker container exec c2 ping 172.17.0.3
```

The request hangs until it times out (press CTRL+C to give up early if you don't want to wait for the timeout). Different docker networks are firewalled from each other by default; dump your iptables rules and look for lines similar to the following:

```
[centos@node-1 ~]$ sudo iptables-save

...
-A DOCKER-ISOLATION-STAGE-1 -i br-dfda80f70ea5 ! -o br-dfda80f70ea5 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o br-dfda80f70ea5 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
...
```

The first line above forwards traffic originating from br-dfda80f70ea5 (that's your custom bridge) but destined somewhere else to the stage 2 isolation chain, where if it is destined for the docker0 bridge, it gets dropped, preventing traffic from going from one bridge to another.

4.3 Forwarding a Host Port to a Container

1. Start an nginx container with a port exposure:

   ```
   [centos@node-1 ~]$ docker container run -d -p 8000:80 nginx
   ```

   This syntax asks docker to forward all traffic arriving on port 8000 of the host's network namespace to port 80 of the container's network namespace. Visit the nginx landing page at <node-1 public IP>:8000.

2. Inspect your iptables rules again to see how docker forwarded this traffic:

   ```
   [centos@node-1 ~]$ sudo iptables-save | grep 8000

   -A DOCKER ! -i docker0 -p tcp -m tcp --dport 8000 -j DNAT --to-destination 172.17.0.4:80
   ```

   Inspect your default bridge network to find the IP of your nginx container; you should find that it matches the IP in the network address translation rule above, which states that any traffic arriving on port tcp/8000 on the host should be network address translated to 172.17.0.4:80 - the IP of our nginx container and the port we exposed with the -p 8000:80 flag when we created this container.

3. Clean up your containers and networks:

   ```
   [centos@node-1 ~]$ docker container rm -f $(docker container ls -aq)
   [centos@node-1 ~]$ docker network rm my_bridge
   ```

4.4 Conclusion

In this demo, we stepped through the basic behavior of docker software defined bridge networks, and looked at the technology underpinning them such as linux bridges, veth connections, and iptables rules. From a practical standpoint, in order for containers to communicate they must be attached to the same docker software defined network (otherwise

they'll be firewalled from each other by the cross-network iptables rules we saw), and in order for containers to resolve each other's name by DNS, they must also be explicitly named upon creation.

5 Instructor Demo: Docker Compose

In this demo, we'll illustrate:

- Starting an app defined in a docker compose file
- Inter-service communication using DNS resolution of service names

5.1 Exploring the Compose File

1. Please download the DockerCoins app from Github and change directory to ~/orchestration-workshop/dockercoins.

```
[centos@node-0 ~]$ git clone -b ee2.1 \
    https://github.com/docker-training/orchestration-workshop.git
[centos@node-0 ~]$ cd ~/orchestration-workshop/dockercoins
```

2. Let's take a quick look at our Compose file for Dockercoins:

```
version: "3.1"

services:
rng:
    image: training/dockercoins-rng:1.0
    networks:
    - dockercoins
    ports:
    - "8001:80"

hasher:
    image: training/dockercoins-hasher:1.0
    networks:
    - dockercoins
    ports:
    - "8002:80"

webui:
    image: training/dockercoins-webui:1.0
    networks:
    - dockercoins
    ports:
    - "8000:80"

redis:
    image: redis
    networks:
    - dockercoins

worker:
    image: training/dockercoins-worker:1.0
    networks:
    - dockercoins

networks:
    dockercoins:
```


This Compose file contains 5 services, along with a bridge network.

3. When we start the app, we will see the service images getting downloaded one at a time:

```
[centos@node-0 dockercoins]$ docker-compose up -d
```

4. After starting, the images required for this app have been downloaded:

```
[centos@node-0 dockercoins]$ docker image ls | grep "dockercoins"
```

5. Make sure the services are up and running, as is the dedicated network:

```
[centos@node-0 dockercoins]$ docker-compose ps
[centos@node-0 dockercoins]$ docker network ls
```

6. If everyting is up, visit your app at <node-0 public IP>:8000 to see Dockercoins in action.

5.2 Communicating Between Containers

1. In this section, we'll demonstrate that containers created as part of a service in a Compose file are able to communicate with containers belonging to other services using just their service names. Let's start by listing our DockerCoins containers:

```
[centos@node-0 dockercoins]$ docker container ls | grep 'dockercoins'
```

2. Now, connect into one container; let's pick `webui`:

```
[centos@node-0 dockercoins]$ docker container exec -it <Container ID> bash
```

3. From within the container, ping `rng` by name:

```
[root@<Container ID>]# ping rng
```

Logs should be outputted resembling this:

```
PING rng (172.18.0.5) 56(84) bytes of data.
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=1 ttl=64 time=0.108
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=2 ttl=64 time=0.049
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=3 ttl=64 time=0.073
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=4 ttl=64 time=0.067
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=5 ttl=64 time=0.057
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=6 ttl=64 time=0.074
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=7 ttl=64 time=0.052
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=8 ttl=64 time=0.057
64 bytes from dockercoins_rng_1.dockercoins_dockercoins (172.18.0.5): icmp_seq=9 ttl=64 time=0.080
```

Use `CTRL+C` to terminate the ping. DNS lookup for the services in DockerCoins works because they are all attached to the user-defined `dockercoins` network.

4. After exiting this container, let's navigate to the `worker` folder and take a look at a section of `worker.py`:

```
[centos@node-0 dockercoins]$ cd worker
[centos@node-0 worker]$ cat worker.py

import logging
import os
from redis import Redis
import requests
import time

DEBUG = os.environ.get("DEBUG", "").lower().startswith("y")

log = logging.getLogger(__name__)
```



```
if DEBUG:
    logging.basicConfig(level=logging.DEBUG)
else:
    logging.basicConfig(level=logging.INFO)
    logging.getLogger("requests").setLevel(logging.WARNING)

redis = Redis("redis")

def get_random_bytes():
    r = requests.get("http://rng/32")
    return r.content

def hash_bytes(data):
    r = requests.post("http://hasher/",
                      data=data,
                      headers={"Content-Type": "application/octet-stream"})
    hex_hash = r.text
    return hex_hash
```

As we can see in the last two stanzas, we can direct traffic to a service via a DNS name that exactly matches the service name defined in the docker compose file.

5. Shut down Dockercoins and clean up its resources:

```
[centos@node-0 dockercoins]$ docker-compose down
```

5.3 Conclusion

In this exercise, we stood up an application using Docker Compose. The most important new idea here is the notion of Docker Services, which are collections of identically configured containers. Docker Service names are resolvable by DNS, so that we can write application logic designed to communicate service to service; all service discovery and load balancing between your application's services is abstracted away and handled by Docker.

6 Instructor Demo: Self-Healing Swarm

In this demo, we'll illustrate:

- Setting up a swarm
- How swarm makes basic scheduling decisions
- Actions swarm takes to self-heal a docker service

6.1 Setting Up a Swarm

1. Start by making sure no containers are running on any of your nodes:

```
[centos@node-0 ~]$ docker container rm -f $(docker container ls -aq)
[centos@node-1 ~]$ docker container rm -f $(docker container ls -aq)
[centos@node-2 ~]$ docker container rm -f $(docker container ls -aq)
[centos@node-3 ~]$ docker container rm -f $(docker container ls -aq)
```

2. Initialize a swarm on one node:

```
[centos@node-0 ~]$ docker swarm init

Swarm initialized: current node (xyz) is now a manager.
```



```
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

3. List the nodes in your swarm:

```
[centos@node-0 ~]$ docker node ls

ID      HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS
xyz *   node-0     Ready    Active         Leader
```

4. Add some workers to your swarm by cutting and pasting the `docker swarm join...` token Docker provided in step 2 above:

```
[centos@node-1 ~]$ docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377
[centos@node-2 ~]$ docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377
[centos@node-3 ~]$ docker swarm join --token SWMTKN-1-0s96... 10.10.1.40:2377
```

   Each node should report `This node joined a swarm as a worker.` after joining.

5. Back on your first node, list your swarm members again:

```
[centos@node-0 ~]$ docker node ls

ID      HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS
ghi     node-3     Ready    Active
def     node-2     Ready    Active
abc     node-1     Ready    Active
xyz *   node-0     Ready    Active         Leader
```

   You have a four-member swarm, ready to accept workloads.

6.2 Scheduling Workload

1. Create a service on your swarm:

```
[centos@node-0 ~]$ docker service create \
    --replicas 4 \
    --name service-demo \
    centos:7 ping 8.8.8.8
```

2. List what processes have been started for your service:

```
[centos@node-0 ~]$ docker service ps service-demo

ID            NAME            IMAGE     NODE    DESIRED STATE  CURRENT STATE
g3dimc0nkoha  service-demo.1  centos:7  node-3  Running        Running 18 seconds ago
e7d7sy5saqqo  service-demo.2  centos:7  node-0  Running        Running 18 seconds ago
wv0culf6w8m6  service-demo.3  centos:7  node-1  Running        Running 18 seconds ago
ty35gss71mpf  service-demo.4  centos:7  node-2  Running        Running 18 seconds ago
```

   Our service has scheduled four tasks, one on each node in our cluster; by default, swarm tries to spread tasks out evenly across hosts, but much more sophisticated scheduling controls are also available.

6.3 Maintaining Desired State

1. Connect to `node-1`, and list the containers running there:


```
[centos@node-1 ~]$ docker container ls
CONTAINER ID  IMAGE     COMMAND         CREATED        STATUS        NAMES
5b5f77c67eff  centos:7  "ping 8.8.8.8"  4 minutes ago  Up 4 minutes  service-demo.3.wv0cul...
```

Note the container's name indicates the service it belongs to.

2. Let's simulate a container crash, by killing off this container:

```
[centos@node-1 ~]$ docker container rm -f <container ID>
```

Back on our swarm manager, list the processes running for our `service-demo` service again:

```
[centos@node-0 ~]$ docker service ps service-demo

ID            NAME                IMAGE     NODE    DESIRED STATE  CURRENT STATE
g3dimc0nkoha  service-demo.1       centos:7  node-3  Running        Running 6 minutes ago
e7d7sy5saqqo  service-demo.2       centos:7  node-0  Running        Running 6 minutes ago
u7l8vf2hqw0z  service-demo.3       centos:7  node-1  Running        Running 3 seconds ago
wv0culf6w8m6  \_ service-demo.3    centos:7  node-1  Shutdown       Failed 3 seconds ago
ty35gss71mpf  service-demo.4       centos:7  node-2  Running        Running 6 minutes ago
```

Swarm has automatically started a replacement container for the one you killed on `node-1`. Go back over to `node-1`, and do `docker container ls` again; you'll see a new container for this service up and running.

3. Next, let's simulate a complete node failure by rebooting one of our nodes:

```
[centos@node-3 ~]$ sudo reboot now
```

4. Back on your swarm manager, check your service containers again:

```
[centos@node-0 ~]$ docker service ps service-demo
ID            NAME                IMAGE     NODE    DESIRED STATE  CURRENT STATE
ralm6irhj6vu  service-demo.1       centos:7  node-0  Running        Running 19 seconds ago
g3dimc0nkoha  \_ service-demo.1    centos:7  node-3  Shutdown       Running 38 seconds ago
e7d7sy5saqqo  service-demo.2       centos:7  node-0  Running        Running 12 minutes ago
u7l8vf2hqw0z  service-demo.3       centos:7  node-1  Running        Running 5 minutes ago
wv0culf6w8m6  \_ service-demo.3    centos:7  node-1  Shutdown       Failed 5 minutes ago
ty35gss71mpf  service-demo.4       centos:7  node-2  Running        Running 12 minutes ago
```

The process on `node-3` has been scheduled for `SHUTDOWN` when the swarm manager lost connection to that node, and meanwhile the workload has been rescheduled onto `node-0` in this case. When `node-3` comes back up and rejoins the swarm, its container will be confirmed to be in the `SHUTDOWN` state, and reconciliation is complete.

5. Remove your `service-demo`:

```
[centos@node-0 ~]$ docker service rm service-demo
```

All tasks and containers will be removed.

6.4 Conclusion

One of the great advantages of the portability of containers is that we can imagine orchestrators like Swarm which can schedule and re-schedule workloads across an entire datacenter, such that if a given node fails, all its workload can be automatically moved to another host with available resources. In the above example, we saw the most basic examples of this 'reconciliation loop' that swarm provides: the swarm manager is constantly monitoring all the containers it has scheduled, and replaces them if they fail or their hosts become unreachable, completely automatically.

7 Instructor Demo: Kubernetes Basics

In this demo, we'll illustrate:

- Setting up a Kubernetes cluster with one master and two nodes
- Scheduling a pod, including the effect of taints on scheduling
- Namespaces shared by containers in a pod

7.1 Initializing Kubernetes

1. On `node-0`, initialize the cluster with `kubeadm`:

```
[centos@node-0 ~]$ sudo kubeadm init --pod-network-cidr=192.168.0.0/16 \
    --ignore-preflight-errors=SystemVerification
```

   If successful, the output will end with a join command:

```
...
You can now join any number of machines by running the following on each node
as root:

  kubeadm join 10.10.29.54:6443 --token wdytg5.q1w1f4dau7u6wk11 --discovery-token-ca-cert-hash sha2
```

2. To start using you cluster, you need to run:

```
[centos@node-0 ~]$ mkdir -p $HOME/.kube
[centos@node-0 ~]$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[centos@node-0 ~]$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

3. List all your nodes in the cluster:

```
[centos@node-0 ~]$ kubectl get nodes
```

   Which should output something like:

```
NAME      STATUS     ROLES     AGE       VERSION
node-0    NotReady   master    2h        v1.11.1
```

   The `NotReady` status indicates that we must install a network for our cluster.

4. Let's install the Calico network driver:

```
[centos@node-0 ~]$ kubectl apply -f https://bit.ly/2v9yaaV
```

5. After a moment, if we list nodes again, ours should be ready:

```
[centos@node-0 ~]$ kubectl get nodes -w
NAME      STATUS     ROLES     AGE       VERSION
node-0    NotReady   master    1m        v1.11.1
node-0    NotReady   master    1m        v1.11.1
node-0    NotReady   master    1m        v1.11.1
node-0    Ready     master    2m        v1.11.1
node-0    Ready     master    2m        v1.11.1
```

7.2 Exploring Kubernetes Scheduling

1. Let's create a `demo-pod.yaml` file on `node-0` after enabling Kubernetes on this single node:

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
```



```
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx
  - name: mydemo
    image: centos:7
    command: ["ping", "8.8.8.8"]
```

2. Deploy the pod:

```
[centos@node-0 ~]$ kubectl create -f demo-pod.yaml
```

3. Check to see if the pod is running:

```
[centos@node-0 ~]$ kubectl get pod demo-pod

NAME       READY     STATUS    RESTARTS   AGE
demo-pod   0/2       Pending   0          7s
```

The status should be stuck in pending. Why is that?

4. Let's attempt to troubleshoot by obtaining some information about the pod:

```
[centos@node-0 ~]$ kubectl describe pod demo-pod
```

In the bottom section titled `Events:`, we should see something like this:

```
...
Events:
  Type     Reason            ...  Message
  ----     ------            ...  -------
  Warning  FailedScheduling  ...  0/1 nodes are available: 1 node(s)
                                  had taints that the pod didn't tolerate.
```

Note how it states that the one node in your cluster has a taint, which is Kubernetes's way of saying there's a reason you might not want to schedule pods there.

5. Get some state and config information about your single kubernetes node:

```
[centos@node-0 ~]$ kubectl describe nodes
```

If we scroll a little, we should see a field titled `Taints`, and it should say something like:

```
Taints:  node-role.kubernetes.io/master:NoSchedule
```

By default, Kubernetes masters carry a taint that disallows scheduling pods on them. While this can be overridden, it is best practice to not allow pods to get scheduled on master nodes, in order to ensure the stability of your cluster.

6. Execute the join command you found above when initializing Kubernetes on `node-1` and `node-2` (you'll need to add `sudo` to the start, and `--ignore-preflight-errors=SystemVerification` to the end), and then check the status back on `node-0`:

```
[centos@node-1 ~]$ sudo kubeadm join...--ignore-preflight-errors=SystemVerification
[centos@node-2 ~]$ sudo kubeadm join...--ignore-preflight-errors=SystemVerification
[centos@node-0 ~]$ kubectl get nodes
```

After a few moments, there should be three nodes listed - all with the `Ready` status.

7. Let's see what system pods are running on our cluster:


```
[centos@node-0 ~]$ kubectl get pods -n kube-system
```

which results in something similar to this:

```
NAME                                       READY     STATUS    RESTARTS   AGE
calico-etcd-pfhj4                          1/1       Running   1          5h
calico-kube-controllers-559c657d6d-ztk8c   1/1       Running   1          5h
calico-node-89k9v                          2/2       Running   0          4h
calico-node-brqxz                          2/2       Running   2          5h
calico-node-zsmh2                          2/2       Running   1          41s
coredns-78fcdf6894-gtj87                   1/1       Running   1          5h
coredns-78fcdf6894-nz2kw                   1/1       Running   1          5h
etcd-node-0                                1/1       Running   1          5h
kube-apiserver-node-0                      1/1       Running   1          5h
kube-controller-manager-node-0             1/1       Running   1          5h
kube-proxy-qxfzt                           1/1       Running   0          41s
kube-proxy-vgrtm                           1/1       Running   0          4h
kube-proxy-ws2z5                           1/1       Running   0          5h
kube-scheduler-node-0                      1/1       Running   1          5h
```

We can see the pods running on the master: etcd, api-server, controller manager and scheduler, as well as calico and DNS infrastructure pods deployed when we installed calico.

8. Finally, let's check the status of our demo pod now:

```
[centos@node-0 ~]$ kubectl get pod demo-pod
```

Everything should be working correctly with 2/2 containers in the pod running, now that there are un-tainted nodes for the pod to get scheduled on.

7.3 Exploring Containers in a Pod

1. Let's interact with the centos container running in demo-pod by getting a shell in it:

```
[centos@node-0 ~]$ kubectl exec -it -c mydemo demo-pod -- /bin/bash
```

Try listing the processes in this container:

```
[root@demo-pod /]# ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  24860  1992 ?        Ss   14:48   0:00 ping 8.8.8.8
root         5  0.0  0.0  11832  3036 pts/0    Ss   14:48   0:00 /bin/bash
root        20  0.0  0.0  51720  3508 pts/0    R+   14:48   0:00 ps -aux
```

We can see the ping process we containerized in our yaml file running as PID 1 inside this container, just like we saw for plain containers.

2. Try reaching Nginx:

```
[root@demo-pod /]# curl localhost:80
```

You should see the HTML for the default nginx landing page. Notice the difference here from a regular container; we were able to reach our nginx deployment from our centos container on a port on localhost. The nginx and centos containers share a network namespace and therefore all their ports, since they are part of the same pod.

7.4 Conclusion

In this demo, we saw two scheduling innovations Kubernetes offers: taints, which provide 'anti-affinity', or reasons not to schedule a pod on a given node; and pods, which are groups of containers that are always scheduled on the same

node, and share network, IPC and hostname namespaces. These are both examples of Kubernetes's highly expressive scheduling, and are both difficult to reproduce with the simpler scheduling offered by Swarm.

